UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

PARALLEL ALGORITHMS FOR COUNTING PROBLEMS ON GRAPHS USING GRAPHICS PROCESSING UNITS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

AMLAN CHATTERJEE
Norman, Oklahoma
2014

PARALLEL ALGORITHMS FOR COUNTING PROBLEMS ON GRAPHS USING GRAPHICS PROCESSING UNITS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Dr. Sridhar Radhakrishnan, Chair

Dr. Suleyman Karabuk

Dr. John K. Antonio

Dr. Sudarshan Dhall

Dr. S. Lakshmivarahan

To Dad

Acknowledgements

First, I would like to sincerely thank my research advisor, Professor Sridhar Radhakrishnan, for inculcating in me a zeal to explore the intricacies of computer science, for encouraging me to work in the forefront of technological advancements, for his guidance and unconditional support during all these years of graduate school, and for being a excellent mentor; I am grateful for having the opportunity to work with him. I would also like to thank Dr. Suleyman Karabuk for being in my committee as the external member and providing valuable insights in many research meetings right from the beginning of this work. It has also been a wonderful learning experience while working with Dr. John Antonio on many of the research problems. Also other members of my dissertation committee Dr. Sudarshan Dhall and Dr. S. Lakshmivarahan have been always encouraging. In addition I had the privilege of working with Dr. Deborah Trytten and Dr. Changwook Kim; teachers from whom I learned the art of communication and conveying ideas.

The School of Computer Science at OU has provided me with great learning experiences and I leave with better education, insight and more surety as an individual and professional in the field. I would like to thank all the faculty and the wonderful staff, specially Barbara, Chryl, Emily, Virginie, Jonathan and Jim Summers, who make this a great learning center.

I also express my gratitude to fellow graduate students and colleagues who made every moment at OU worthwhile. I would like to thank all people associated with Dr. Radhakrishnan's research group, Chandrika Satyavolu, Yuh-Rong Chen, Asif M. Adnan, Khondker Shajadul Hasan, Sourabh Joshi, Michael Nelson, Yiming Xu, Wei Guo, Ashwin Badri, Mahendran Veeramani. Also thanks to some great people whom

I have had the privilege to know and associate with during my time at the University of Oklahoma: Harshvardhan Singh, Raghav Pant, Nehul Shah, Syed Naveed Hussain Shah, Shohrab Hossain, Rahul Shukla and all the CSGSA committee members. I would also like to thank my friends from Buffalo, Debarshi Indra, Debmalya Das Sharma, Divya Shankar; my college friend Niladri Kundu, school-mate Ambarish Banerjee and senior from school Swapnoneel Roy.

I would also wholeheartedly like to acknowledge the love and support of my family: my father, late Mr. Ranjit Kumar Chatterjee, my mother Mrs. Subhra Chatterjee; my aunt Ms. Kalpana Mukherjee; my brother, Dr. Anindya Chatterjee and sister-in-law Dr. Allison Chatterjee; my in-laws, Mr. Murari Mohan Banerjee and Mrs. Santwana Banerjee; who have been a constant inspiration, and also my grandparents, aunts, uncles, and cousins everywhere; I could not have accomplished this without you.

Finally, and most importantly, I would like to put in a special note for my wife, Debasmita, whose constant encouragement made this work possible. Without her sacrifice and dedication, this journey would have been incomplete. It is my honor and privilege to find a friend and soul-mate like her, who with her honest feedback and appreciation has made this pursuit of education worthwhile.

Contents

List of Tables

List of Figures

Abstract

The availability of Graphics Processing Units (GPUs) with multicore architecture have enabled parallel computations using extensive multi-threading. Recent advancements in computer hardware have led to the usage of graphics processors for solving general purpose problems. Using GPUs for computation is a highly efficient and low-cost alternative as compared to currently available multicore Central Processing Units (CPUs). Also, in the past decade there has been tremendous growth in the World Wide Web and Online Social Networks. Social networking sites such as Facebook, Twitter and LinkedIn, with millions of users are a huge source of data. These data sets can be used for research in the fields of anthropology, social psychology, economics among others.

Our research focuses on converting real-world problems into graph theoretic problems and using GPUs to solve them. The graph problems that we focus on in our research involve counting the number of subgraphs that satisfy a given property. For example, given a graph $G = (V, E)$ and an integer $k <= |V|$, we provide algorithms to count the number of: a) connected subgraphs of size k; b) cliques of size k; and c) independent sets of size k, and other similar problems. Also, properties that are affected by the dynamic nature of the graphs i.e., addition or removal of edges or nodes, for example change in the number of triangles and connected components in the graph, are also studied.

Sequential access to global memory and contention at the size-limited shared memory have been main impediments to fully exploiting potential performance in GPUs. Therefore, we propose novel memory storage and retrieval methods, based on using search techniques on graphs and converting it into trees, that enable parallel graph

computations to overcome the above issues. We also analyze and utilize primitives such as memory access coalescing and avoiding partition camping that offset the increase in access latency of using a slower but larger global memory. In addition, we introduce graph compression techniques that further reduce memory requirements and overheads. Our experimental results for the GPU implementation show a significant speedup over the CPU counterpart for the problems described above.

Chapter 1

Introduction

Majority of real-world data, including those that are being generated online, can be represented as graphs. Representing data as graphs has two significant advantages: on one hand visual representation can convey knowledge about the data which otherwise could have been difficult to interpret, and on the other hand, graph theory being a mature and well-studied discipline can be leveraged by using its algorithms and results to study the data being considered. Analyzing this huge amount of information, gathered from the graphs, leads to potential insights that are of interest to various disciplines spanning across both academia and industry. The volume of data being processed pose new challenges in terms of storage and computation time, and utilizing the latest advancements in hardware architecture can help address the same.

1.1 Overview

The continuous growth and availability of huge graphs for modeling online social networks, World Wide Web and biological systems has rekindled interests in their analysis. Social networking sites such as Facebook with 1.3 billion users (Facebook Statistics, 2014), Twitter with 271 million users (Twitter Statistics, 2014) and LinkedIn with over 300 million users (LinkedIn Press Center, 2014) are a huge source of data for research in the fields of anthropology, social psychology, economics and others. Understanding the structure of OSNs help in improving Internet search, advertising, and even mitigating against spamming and other security issues like Sybil attack (Yu et al., 2006). Finding subgraphs that satisfy a specific property in such networks,

and solving other problems based on locality information is therefore of practical significance. Therefore, it is relevant to study these data and perform analysis on the same. However, due to the huge amount of computation involved, it is impractical to analyze very large graphs with a single Central Processing Unit (CPU), even if multi-threading is employed.

Recent advancements in computer hardware have led to the usage of graphics processors for solving general purpose problems. Compute Unified Device Architecture (CUDA) from Nvidia (NVIDIA Corporation, 2010) and Accelerated Parallel Processing (APP) Technology from AMD are interfaces for modern Graphical Processing Units (GPUs) that enable the use of graphics cards as powerful co-processors. These systems enable acceleration of various algorithms including those involving graphs (Harish & Narayanan, 2007).

Solving general-purpose problems on GPUs is a highly efficient and low-cost alternative to currently available multicore CPUs. Available hardware and the potential for massive multi-threading has shifted the focus from using a GPU as a graphics renderer to a powerful co-processor. Among other problems that have achieved speedup from being solved on the GPU, analysis of graphs, which is an inherent operation in many real world applications, with combinatorially explosive number of computations has the potential to exploit the available architecture of GPUs.

1.2 Motivation

The motivation of the research and study of this dissertation comes from two important aspects. One is the need to develop algorithms and programs that can take advantage of the multicore architecture and exploit the available hardware in both CPUs and GPUs. The other is the availability of huge data sets generated from various resources that contain a plethora of information that needs to be processed and

analyzed for valuable insights. A lot of analysis is done on real-world data and most of it is done sequentially or using a few threads to maximize the use of multicore CPUs. However, in many cases the data that is being computed on is independent or has limited dependencies, and there is a lot of potential to do computations in parallel. Gene Amdahl proposed an estimate on the upper bound to the amount of parallelization that can be incorporated in an algorithm, known as Amdahl's law. It says that, in general, if a fraction α of an application can be run in parallel and the rest must run serially, the speedup is at most $\frac{1}{(1-\alpha)}$. Therefore, it is important to identify and convert the parts of an algorithm that can benefit from being transferred into equivalent parallel counterparts.

1.2.1 Advancements in computer hardware

The processing power of computers have improved steadily over the last few decades. This was in conjunction with the observation made by Gordon E. Moore, which is now commonly referred to as Moore's law. Moore's law predicts that the number of transistors in a dense integrated circuit would double every two years. Since the number of available transistors are a measure of the speed of the computer, the simplified version of the observation states that processor speeds would double every two years. However, with the size of transistors reaching the lower end and increasing power dissipation from the chips, there would be changes to the rate of this development. To counter this problem, CPUs with dual-cores and quad-cores have already come into existence. This alleviates the problem of packing transistors into a single chip by using more than one of the same. It is also common nowadays to have computers with multiple multi-core CPUs. However, this still does not provide too many compute cores and is an expensive option.

In recent times, the hardware architecture of the GPUs have also evolved significantly. With a collection of a number of streaming multi-processors, each of which can

contains a few hundred cores, the multicore architecture advancement is now spearheaded by the GPUs. With the inclusion of multicore GPUs in most desktop and laptop computers, huge processing power has now become available for widespread use. Also, the low-cost of these devices as compared to multi-core CPUs make it an affordable option. Compute Unified Device Architecture (CUDA) from Nvidia (NVIDIA Corporation, 2010) and Accelerated Parallel Processing (APP) Technology from AMD (Advanced Micro Devices, 2013) are interfaces for modern Graphical Processing Units (GPUs) that enable the use of graphics cards to solve general purpose problems. This has led to referring modern GPUs as General Purpose Graphics Processing Units or GPGPUs.

1.2.2 Availability of large data sets

Variety of large data sets containing information about different real-world entities and their interactions are widely available. Data can be either static or dynamic depending on whether it changes over time. For static data, performing a one-time analysis is sufficient to study specific characteristics and properties. However, for dynamic data, performing repeated analysis with the passage of time is required to incorporate the recent trends.

Data sets can represent various entities such as online social networks, communication networks, citation networks, collaboration networks, web graphs, road networks, online reviews, sales, airline routes, protein interactions, weather patterns etc. among others. Majority of these data form graphs or networks that are huge in size. Also, the rate at which data is generated and added to the existing sets is enormous.

Data from various sources contain different patterns that of interest and can shed light into greater understanding of the inner workings of the respective domains they belong to. Therefore, it is a significant step to be able to perform relevant analysis on the data and extract information from the same.

1.2.3 Data Analysis Applications

As discussed in the previous sub-section, data exists in various forms and analyzing it is relevant for different purposes. This sub-section looks into the different applications based on analysis of data. Depending on the domain the data is generated from, there can be various applications and uses for the information extracted from the analyzed data.

For example, in the case of online social networks, the interaction and activities among users provide vital information that can be analyzed for use in advertising, improving user experience, security, etc. Sales data, including both online sales and those in the stores, can provide information to the merchants and suppliers about products that are in demand during specific time periods. Weather predictions for a specific region is dependent on patterns observed in the adjoining areas; with huge volumes of changing values that can be calculated quickly, predictions about inclement weather patterns improve in accuracy. Analysis on road networks help find the major points of intersection and the roads that are important to maintain connectivity throughout a region. Construction and other maintenance work can change the graph structure where certain connections or edges become non-existent due to inaccessibility; also expansion work to reduce load on certain roads can be found by studying the effects of adding potential connectors between junctions. Data from biological networks including protein interactions can be analyzed to find specific patterns that are of significant research value.

Therefore, it can be inferred that there are various important applications that are dependent on the correct and timely analysis of huge amount of data.

1.3 General purpose computing using GPUs

CUDA enabled GPUs provide high performance computing on the desktop, which is a low-cost and low-power alternative to conventional super-computing. GPUs have several multi-processors, each of which has multiple cores. Therefore, GPUs are suitable for highly parallel and multi-threaded applications.

GPUs are primarily used to render graphics on the screens. Therefore, the architecture is designed to help calculate pixel values in a fast manner. However, previously GPUs have also been used to solve problems other than those related to graphics rendering. The approach involved converting the specific problem into a graphics problem, solving it using the GPU and then converting the results back by mapping it back to the original problem.

However, as the process of transferring a problem to a different domain and back required significant effort, it was detrimental to GPUs being used to solve general purpose problems. With the advent of CUDA programming model, general purpose problems can be solved using GPUs in their original form, thereby removing the overhead of converting problems to an equivalent graphics version. Also, since the learning curve is low and the programming syntax is similar to existing languages, variety of research and academic fields have taken advantage of the computing resources provided by GPUs.

Some of the domains that benefit from using GPUs are bio-informatics, computational chemistry, computational finance, computational fluid dynamics, computational structural mechanics, data science, defense, electronic design automation, imaging & computer vision, machine learning, medical imaging, numerical analytics, weather prediction.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses about GPU architecture and CUDA programming & memory model. The memory hierarchy, access patterns and optimization methodologies are also discussed here. Techniques for efficiently storing graphs on GPUs are introduced in Chapter 3; a number of naïve and advanced data structures are studied and compared. Counting problems related to graphs are analyzed in Chapter 4; we discuss methods to count the number of sub-graphs in a given graph that satisfy certain criteria. Chapter 5 discusses about analysis on large data sets pertaining to real-world data. We study the properties of graphs representing online social networks and road networks, and introduce algorithms to study the effects of changes in graph data. Data compression algorithms that are related to and can be used in graph compression are discussed in Chapter 6. This is essential for storing larger graphs on the GPU memory for efficient computation. Conclusion and future work are given in Chapter 7.

Chapter 2

GPU Architecture & CUDA

Graphics processing units (GPUs) have traditionally been used as co-processors for aiding in displaying content. However, with the advent of GPUs with advanced architecture capable of incorporating multiple cores, often in the hundreds to few thousand, the focus has shifted towards using it for solving general purpose problems. Therefore, understanding the architectural design and the programming capabilities of these devices is relevant. In this chapter we study the GPU architecture and a parallel computing platform used for programming the same.

2.1 Introduction

Central Processing Units (CPUs) have been the traditional compute-engine for solving computational problems. Over the last decade, advancements in computer hardware have led to the usage of graphics processors as accelerators for solving general purpose problems. Using Graphics Processing Units (GPUs) for general purpose computing is referred to as GPGPU computation. Compute Unified Device Architecture (CUDA) from Nvidia (NVIDIA Corporation, 2010), and Accelerated Parallel Processing (APP) Technology from AMD (Advanced Micro Devices, 2013) are interfaces for modern GPUs, which help to use graphics cards as powerful co-processors. The advantages of using GPUs are higher compute performance, usage of less power and lower cost as compared to the corresponding CPU counterparts. Some of the fastest super-computers in the world, like the Tianhe-IA, are powered by using Nvidia GPUs (Sulewski et al., 2011).

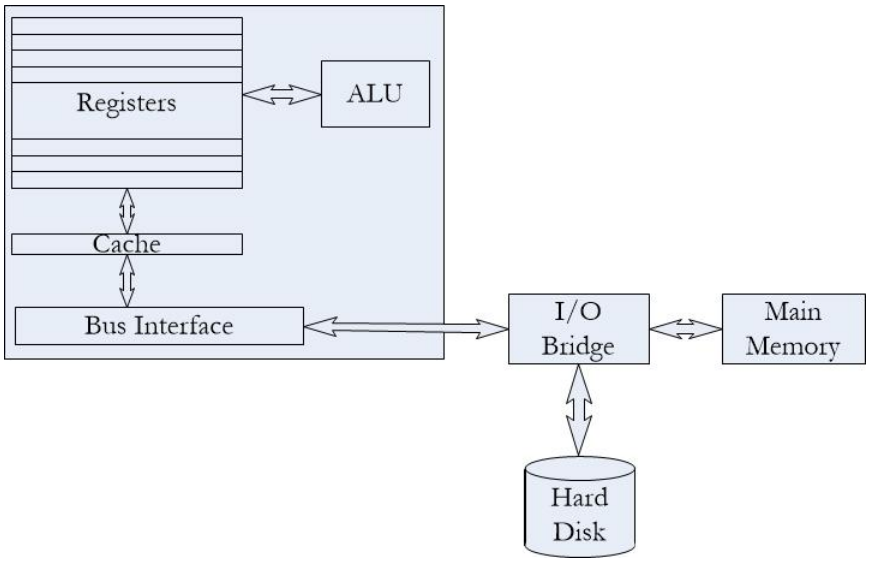


Figure 2.1: Architecture of a single core CPU

2.2 GPU Architecture

Computers have traditionally had single core processors cast on a CPU chip. The core is comprised of a single set of registers with a corresponding Arithmetic Logic Unit (ALU). Input-output to this unit is done using the bus interface. Other than the main memory, which is accessible using the memory bus, the CPU chip also makes use of available on-chip cache apart from the registers for storing temporary data. This memory hierarchy, consisting of the registers, different levels of cache, and the main memory determines the performance of the CPU while executing data intensive applications by reducing the latency introduced due to accessing of data from main memory or even the external disk drives. The architecture of such a device is shown in Fig. 2.1.

In case of multicore CPUs, the chip consists of multiple sets of ALUs and registers. Each set of an ALU and registers is defined as a core for the computer. Fig. 2.2 shows a chip consisting of 3 cores.

Other than multicore CPUs, GPUs also make use of multicore architecture for

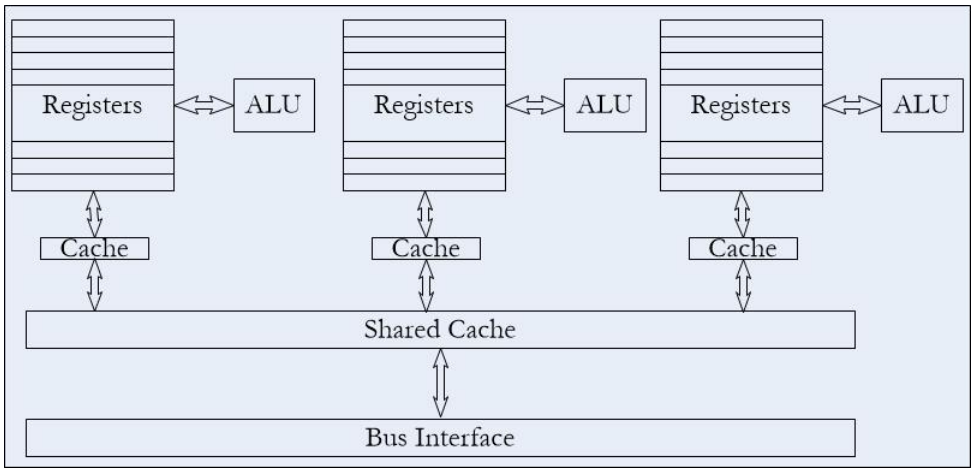


Figure 2.2: Architecture of a multi-core CPU

computations. CUDA, acronym for Compute Unified Device Architecture, developed by NVIDIA Corporation, provides a programming and memory model for GPUs to help those perform as general purpose graphics processing units (GPGPUs). In the CPU-GPU heterogeneous environment, the GPU is referred to as the "device" and the CPU to which it is connected is called the "host". The GPU can be controlled and accessed by programs executing on the CPU and data can be transferred to the memory of the device to delegate specific tasks to be performed on it. Earlier, GPUs were specifically used to solve programs that belonged to the graphics domain. One way to utilize the computation power of the GPUs is to convert general programs into equivalent graphics problem, and then solve those problems instead. But this approach is complicated and there is a lot of conversion overhead involved. CUDA enabled GPUs on the other hand allows users to directly execute programs and solve general problems in the original form. The CUDA API (Application Programming Interface) documents all the details as to how the programs that are executed on the CPU can transfer or delegate a part of the program to be executed on the GPU.

CUDA provides a large number of threads that can be executed simultaneously on the cores of the device. To be able to maximize the utilization of the available hardware, it is necessary to have parallel versions of the problems that are expected

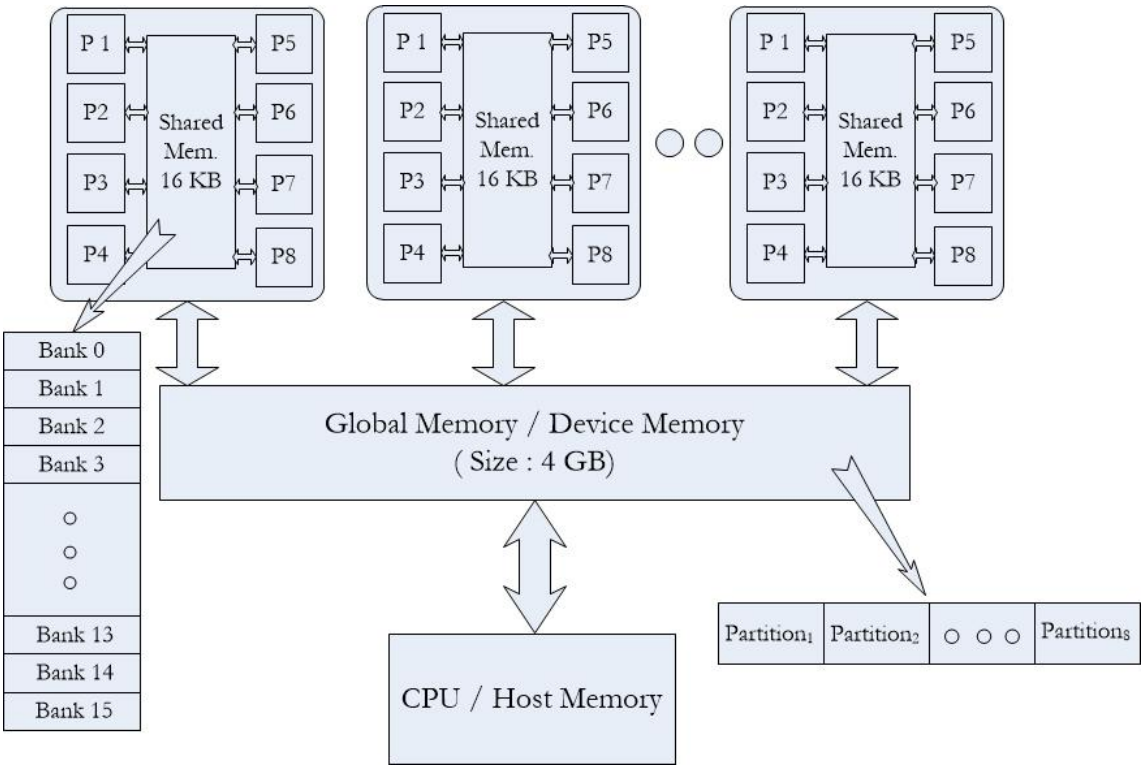


Figure 2.3: Architecture of a GPU (C1060)

to have performance gain by solving on multicore architecture. Therefore, developing parallel versions of both basic and advanced algorithms for different domains is relevant. Significant performance gains can be achieved by executing highly multithreaded applications on the CUDA enabled GPUs. But, along with this opportunity to achieve significant speedup, there is the challenge of minimizing the latency introduced due to simultaneous data access from the memory. As in CPUs, there is a hierarchy of memory units for the GPUs. Similar to the registers, L1, L2 and other levels of cache which are used to hide the latency introduced by the memory accesses, the GPUs have their own registers and shared memory, which are on-chip and can be used by the GPU for fast execution. The architecture of a GPU is shown in Fig. 2.3. The GPU consists of a number of Streaming Multiprocessors (SMs, 30 for C1060), each containing 8 cores, and have access to the on-chip shared memory of that specific SM. All the cores in all the SMs also have access to the device or global

memory. The shared memory is further subdivided into *banks* and the global memory into *partitions*. Memory access from different banks and partitions can be done in parallel.

2.3 CUDA

Compute Unified Device Architecture or CUDA is a programming environment for using GPUs to solve general purpose problems. The CUDA environment consists of the CUDA programming model and the CUDA memory model. The programming model for CUDA extends the C programming language and defines C-like functions, called *kernels*, which are executed in parallel by different CUDA threads. In the following sub-sections we look into the details of CUDA.

2.3.1 CUDA Programming Model

The CUDA programming model provides the basics of how the multi-threaded architecture is designed and functions. The threads are grouped together in *blocks*, which can be either one, two or three dimensional, and each thread within a block is identified by its unique *threadID*. All the blocks of threads, identified by *blockID*, form a one or two dimensional grid. Threads within a block can co-operate among themselves by accessing data through *shared memory* and synchronizing their execution to coordinate memory accesses. Blocks of threads are assigned to the multiprocessors by the CUDA scheduler.

The sequential parts of the code is executed on the "host" i.e., the CPU, and the parallel parts are executed on the "device" i.e., the GPU. Blocks of threads execute the *kernels*. Threads are further divided into execution units called *Warps*. Each *Warp* consists of 32 active threads. But for the execution purpose, the scheduler assigns *half-warps* to the streaming multiprocessors. Therefore, at any instant of time, there

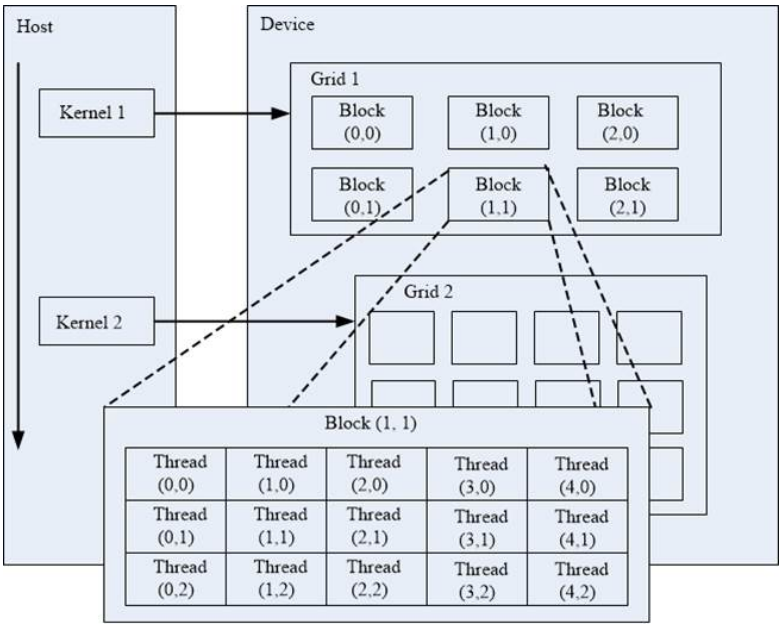


Figure 2.4: CUDA Programming Model (NVIDIA Corporation, 2010)

are 16 active threads on any multiprocessor. So, the total number of threads in a block are divided and grouped together as *Warps*, which in turn are further subdivided into *half-warps*. The blocks of threads in turn are grouped together to form the *Grid*, which can be either one or two dimensional as shown in Fig. 2.4.

Threads from different blocks synchronize among themselves by using the *Global memory*. The maximum number of threads that can reside on a streaming multiprocessor is limited by the hardware resources. There are a limited number of registers with each of the streaming multiprocessors, and with the increase in their usage, the number of threads decreases. The scheduler assigns more than one block of threads to the same multiprocessor if the total number of threads in the blocks combined is less than what can be handled in each of the multiprocessors.

Parallel programming models are classified primarily based on problem decomposition and process interaction. Classification according to problem decomposition can be categorized with respect to task parallelism or data parallelism. CUDA follows the SPMD (Single Program Multiple Data) model, which is similar to the SIMD (Single

Instruction Multiple Data) model. Threads within the same block execute in SIMD fashion, whereas threads of different blocks can execute different portions of the code i.e., the instruction can be different while considering all the streaming multiprocessors, but the program is the same. This is also referred to as SIMT (Single Instruction Multiple Thread) in many cases.

Considering the process interaction, parallel programming models are broadly classified into Shared Memory Model and Distributed Memory Model. In case of Shared Memory Model, the architecture consists of processors accessing a common or shared memory; the PRAM (Parallel Random Access Machine) model is one such example. There are various versions of the PRAM model, the EREW (Exclusive Read Exclusive Write), CREW (Concurrent Read Exclusive Write) and the CRCW (Concurrent Read Concurrent Write), based on memory read and write patterns. In case of the Distributed Memory Model, the architecture consists of a number of processors each with their own local memory, and interaction among the processors are performed by sending messages. Message Passing Interface or MPI is one such example.

The CUDA programming model is actually a combination of both the Shared and Distributed models. CUDA follows the Shared Memory model if the data is stored and accessed using only the global memory. In this case it follows the EREW model. Message Passing Interface is available for GPU clusters using CUDA-Aware MPI thus making is possible to classify it as Distributed memory model.

2.3.2 CUDA Memory Model

There are different levels in the memory hierarchy of the GPUs. Fig. 2.5 shows the different levels of the memory hierarchy available on a GPU. Table 2.1 summarizes some properties of the levels of memory. Data is transferred from the CPU to the GPU and back using memory copy functions. Global memory is predominantly used

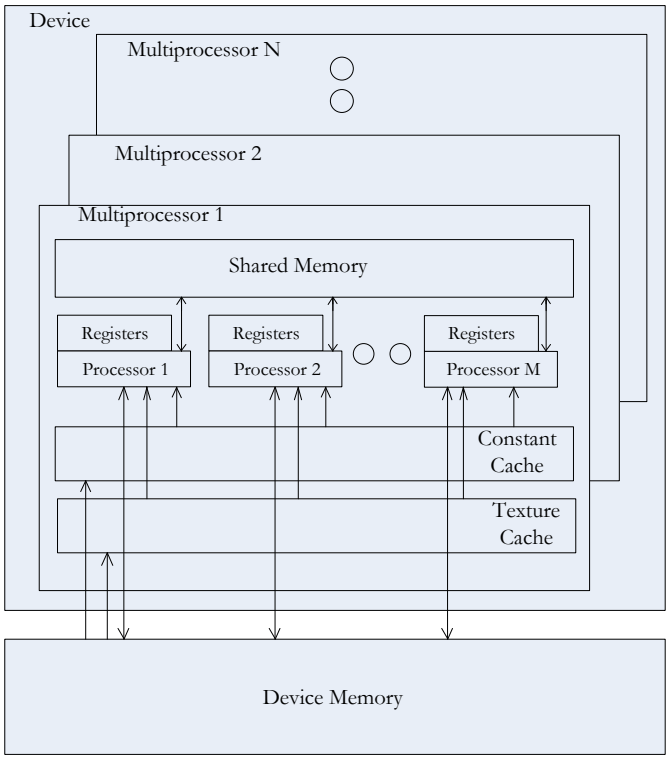


Figure 2.5: GPU memory hierarchy (NVIDIA Corporation, 2010)

to transfer data from the host to device and vice-versa. In the following section we discuss about different characteristics and usage of the memory model.

Memory	Location	Access	Scope	Lifetime
Register	On-chip	R/W	One thread	Thread
Local	Off-chip	R/W	One thread	Thread
Shared	On-chip	R/W	All threads in a block	Block
Global	Off-chip	R/W	All threads + host	Application
Constant	Off-chip	R	All threads + host	Application
Texture	Off-chip	R	All threads + host	Application

Table 2.1: Properties of levels of memory on GPUs (NVIDIA Corporation, 2010)

2.4 Memory Hierarchy, Access Patterns and Optimization Techniques

Memory on the GPU is divided into many different levels. The memory hierarchy of the GPU consists of device memory, shared memory, constant memory, texture memory and registers. The size of each of the above mentioned types of memory varies according to the system, and a comparison of the global and shared memory size for different GPUs is given in Table 2.2. The shared memory is further divided into 16 or 32 banks depending on the system. The global memory is the largest and also has the highest access latency. The on-chip shared memory has significantly faster access compared to the global memory. But, when data is accessed from the same bank, either the same element or different elements in the same bank, then there is a bank conflict leading to a performance loss (the only exception being the case where all the threads access the same element leading to a broadcast.) The constant memory and the texture memory are also located off-chip like the the global memory, but both are read-only memory and data stored in them cannot be modified.

Model #	CUDA cores	Global Memory	Shared Memory	# of Memory Banks
C1060	240	4 GB	16 KB	16
C2050	448	3 GB	48 KB	32
C2070	448	6 GB	48 KB	32

Table 2.2: Architecture Comparison of Different Nvidia GPUs (NVIDIA Corporation, 2010)

Various methods of memory optimization and parallelism management to improve performance of GPUs have been studied (Yang et al., 2010a). The two main considerations for efficient processing on the GPUs is to effectively distribute the workload among the threads and also efficiently utilize the memory hierarchy. Identification of

inefficient workload division or memory accesses in kernels, and reorganizing the code to optimize the memory accesses and maximizing the parallelism using a compiler has been proposed (Yang et al., 2010a). Techniques to improve the performance of the GPUs and performance prediction model has also been studied (Hasan et al., 2014). The focus of the memory optimization techniques can be divided into the following four categories:

- To be able to efficiently access data from global memory and utilize the off-chip memory bandwidth, data item might be vectorized and memory coalescing can be taken advantage of.
- The shared memory has low access latency, and its optimal usage can be ensured by avoiding bank conflicts.
- Workload must be divided among threads in a balanced manner, so that threads scheduled on each of the available multiprocessors does similar amount of work.
- Data accessed from off-chip global memory must avoid partition camping, which is similar to bank conflicts, but with much higher performance penalty.

The above mentioned performance issues are applicable to many-core architectures other than the GPUs. These are universal methodologies, and primitives focusing on high memory bandwidth and balanced workload are relevant to a wide range of architectures.

2.4.1 Shared Memory Vs. Global Memory

As evident from the CUDA programming guide (NVIDIA Corporation, 2010) and related work (Boyer et al., 2008), the shared memory is a lot faster than the global memory. So, for all algorithms, whenever it is possible, the data is stored, accessed and modified from the shared memory. As the shared memory has a capacity of 16

KB (C1060), and different streaming multiprocessors can have different data stored in each of them, the total size of data being processed is limited by that available on the 30 streaming multiprocessors. Therefore, for the C1060 card, the total shared memory available is 480 KB. Although, it must be observed that the entire shared memory might not be available for data storage during the execution of a kernel because of the storage of the kernel parameters and other intrinsic values in it.

2.4.2 Shared Memory Bank Conflicts

The shared memory is of size 16 KB (or 48 KB), and is divided into 16 banks (or 32 banks). Data is stored in consecutive banks, and the width of each bank is 4 bytes.

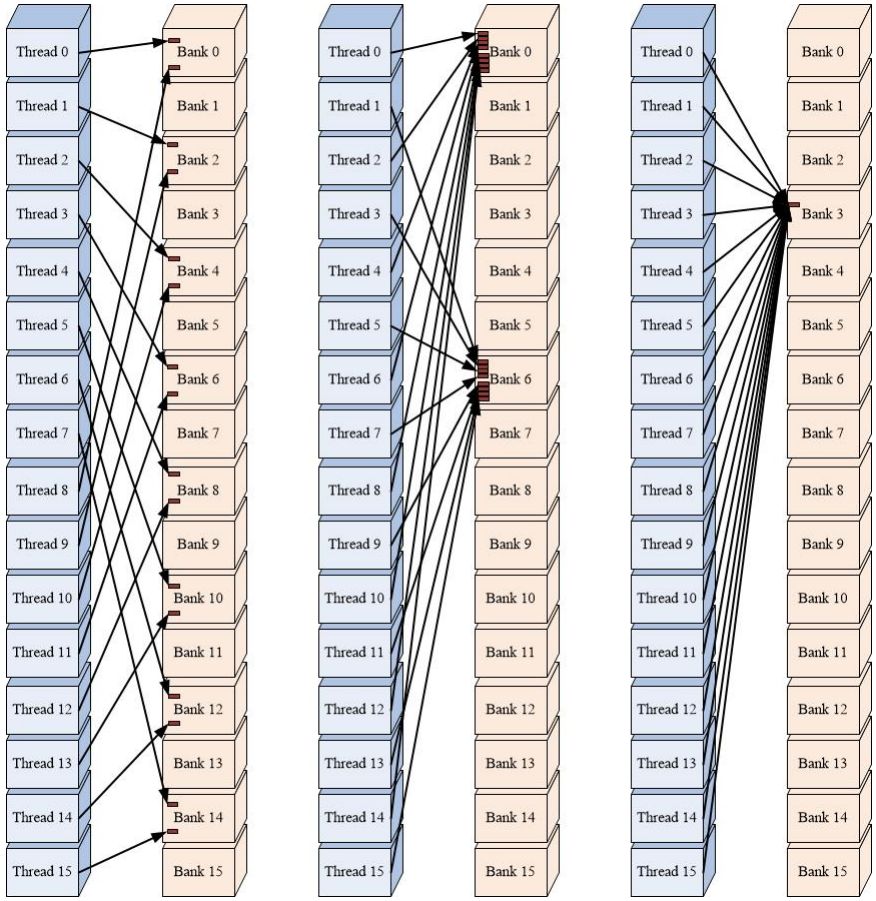


Figure 2.6: Shared Memory Bank Access: (a) No Conflicts, (b) 8-Way Conflict, (c)Broadcast (NVIDIA Corporation, 2010)

Now, all the active threads scheduled on the same streaming multiprocessor can access data from any bank on the shared memory. Data access from different banks take place in parallel. If more than one thread accesses data from the same bank, *bank conflict* occurs, and the accesses become sequential and the memory access latency increases. Fig. 2.6 (a) & (b) shows threads accessing shared memory with no bank conflicts, and also with 8-way bank conflicts. 8-way bank conflicts means 8 threads access data from the same bank at the same time.

In the special case, where all the threads access the same data element from the same bank, the compiler optimizes these memory access operations and issues a broadcast. Therefore, in this case instead of 16-way bank conflict i.e., maximal conflict, there is no conflict at all, and the data is available to all the threads in just a single read operation rather than the required 16 operations. Fig. 2.6 (c) shows threads accessing the same data element in the same bank in the shared memory resulting in the broadcast mechanism taking effect.

But, in the case where even though all the threads access the same bank, but different data elements in the bank, the broadcast mechanism cannot be used. So, maximal bank conflicts occur. The threads are executed on the multiprocessors in groups of 32 referred to as *Warps*. But, only half of the threads in a warp are active at any instant of time. This can be attributed to the fact that there are 16 banks in the shared memory, and if all the threads in a warp are active, then it is guaranteed to cause at least 2-way bank conflicts. Therefore, by using half-warps, there is a chance that the threads can access the shared memory banks without any conflicts.

2.4.3 Global Memory Access Coalescing

In certain cases, where the size of the data required for computation is larger than that of the shared memory, even when using the most efficient data structures, storing and accessing the data from the global memory is required. Due to the increased latency,

as evident from the experimental results in (Boyer et al., 2008), the time required to access data from the global memory is much larger than that from the shared memory. Therefore, to reduce the penalty in execution time while using the global memory as the storage for the data, there are certain primitives available for the Nvidia GPUs that can be used. *Memory Coalescing* is one such mechanism. By taking advantage of memory coalescing, the effects of slower memory accesses can be effectively compensated.

Data from the global memory is accessed in the form of transactions. Therefore, minimizing the number of global memory accesses is equivalent to minimizing the number of transactions.

Let us consider the case where each of the threads in an active *half-warp* (total 16, identifier idx 0 - 15) accesses elements from an array stored in the global memory using a loop. Also, because of the *offset*, let the elements accessed by the above mentioned threads are all in separate segments. Therefore, in this case, each step of the loop would require 16 transactions (one for each thread), as shown in Fig. 2.7.

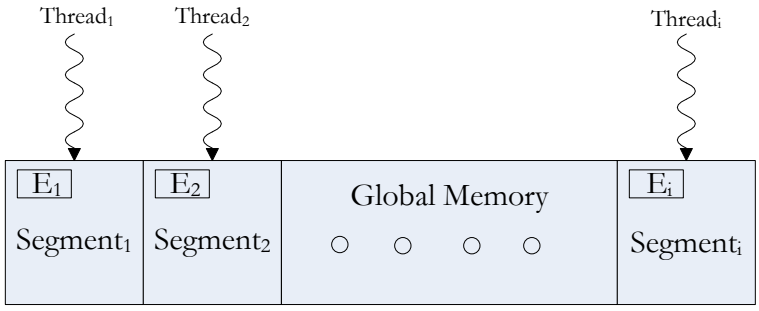


Figure 2.7: Global Memory Access: Maximum Transactions

As mentioned earlier, since transactions are expensive as the time required to access data from the global memory is significant, the number of transactions required to get the data for the threads from the global memory can be reduced by using memory coalescing.

The global memory access by 16 threads (half-warp) is coalesced into a single memory transaction if data accessed by all threads lie in the same segment. Considering the previous example, if the value of *offset* is modified to a value such that all the data elements accessed by the threads belong to the same data segment, then the number of transactions for each step of the loop reduces from 16 to 1, as shown in Fig. 2.8.

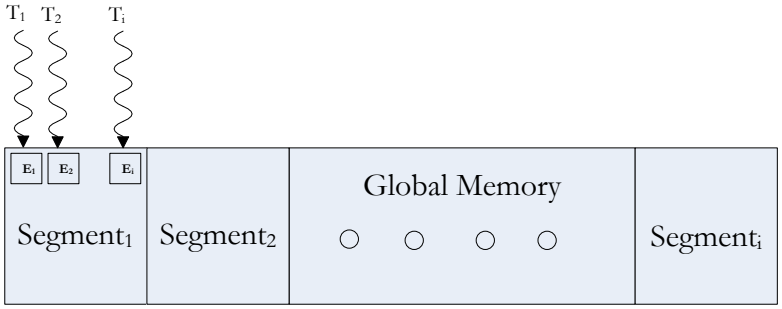


Figure 2.8: Memory Coalescing in Effect: Minimum Transactions

It must be noted that while data elements might be contiguous, they need not be in the same segment. For example, there might be a single transaction required to access 64-bytes of data if it is stored entirely in a single segment. But, for another set of data, accessing a set of 50-bytes might require two transactions if the data is split over two segments in the global memory.

The number of transactions required to access data from the global memory using memory coalescing depends on a number of factors like the compute capability of the system in question and whether the data that is being accessed by the different threads within the active warp is aligned and sequential or non-sequential (NVIDIA Corporation, 2010). A comparison for the different available options is shown in Table 2.3.

From the data available in Table 2.3, it is evident that for CUDA compute capability versions 1.2 and later, accessing non-sequential data is handled in the same

Compute Capability	Access Pattern	Data Size in Bytes	Memory Transactions
1.0	Sequential	128	2
1.1	Sequential	128	2
1.2	Sequential	128	2
1.3	Sequential	128	2
2.0	Sequential	128	1
1.0	Non-sequential	128	32
1.1	Non-sequential	128	32
1.2	Non-sequential	128	2
1.3	Non-sequential	128	2
2.0	Non-sequential	128	1

Table 2.3: Number of Memory Transactions on different GPUs (NVIDIA Corporation, 2010)

manner as sequential data.

2.4.4 Partition Camping

Although memory coalescing is an efficient technique that can be used to offset the slower access to the global memory, there are other factors that can dictate the actual performance benefit achieved. Partition camping is one such mechanism that dominates the outcome of various measures undertaken to access data from the global memory efficiently. Although the shared memory has low access latency, the limiting factor is shared memory bank conflicts. Similarly, from the global memory perspective, memory coalescing is the performance booster while the limiting factor is partition camping. Memory coalescing and partition camping deal with data transfers between the global and on-chip memories, while bank conflicts deal with on-chip shared memory.

The shared memory on the Nvidia systems is divided into 16 (or 32) banks of 32-bit width. Similarly, the global memory is divided into 6 (or 8) partitions on 8-

and 9-series GPUs (or 200- and 10-series GPUs) of 256-byte width, as shown in Fig. 2.9.

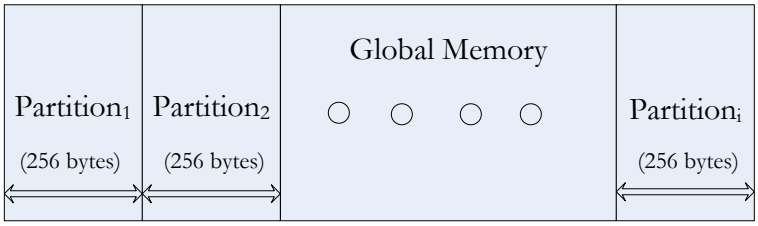


Figure 2.9: Global memory divided into partitions

As evident from the previous discussion, for efficient use of the shared memory, bank conflicts must be avoided. This is ensured by distributing the data accessed by the threads in a half warp into the maximum possible available banks. Therefore, the total time to access the data from the shared memory is indirectly proportional to the number of banks accessed by the threads in the active half-warp, and is given in the form of the following equation

$$\sum_{i=1}^{\beta} T_i \propto \frac{D_e}{\sum_{i=1}^{\beta} B_i} \tag{2.1}$$

where, T_i is the time required for thread i, D_e is the number of data elements accessed by the threads, and $\sum_{i=1}^{\beta} B_i$ is the total number of *distinct* banks accessed by all the threads for computation on the data. Since the *Warp-size* is 32 and *half-warp* is 16, the maximum value of β is 16. In the best-case scenario, when there is no bank conflict, the threads access distinct banks and $\sum_{i=1}^{\beta} B_i = 16$. In the worst-case scenario, when all the threads access the same bank but different elements in it thereby negating the usage of the broadcast primitive available with the compiler, $\sum_{i=1}^{\beta} B_i = 1$.

As the shared memory is divided into banks, similarly the global memory is di-

vided into partitions. For efficient usage of global memory, concurrent accesses by all active warps should be divided evenly amongst partitions. Partition camping occurs when global memory accesses are mapped into a subset of partitions, causing requests to queue up at some partitions while other partitions go unused. While memory coalescing concerns global memory accesses within a half warp, partition camping deals with global memory accesses amongst active half warps. Therefore, it is analogous to shared memory bank conflicts, but on a wider scale. As shown in Equation[2.1] for shared memory bank conflicts, the total time to access the data from the global memory is indirectly proportional to the number of partitions accessed by the threads in all the active half-warps, and is given in the form of the following equation

$$\sum_{i=1}^{\gamma} T_{iw} \propto \frac{\sum_{i=1}^{\gamma} CM_i}{\sum_{i=1}^{\gamma} Part_i} \tag{2.2}$$

where, T_{iw} is the time required for threads in the active warp W_i, CM_i is the total number of coalesced memory access required to access the data elements to be processed by the threads in active warp W_i, and $\sum_{i=1}^{\gamma} Part_i$ is the total number of *distinct* partitions accessed by all the threads in warp W_i for computation on the data, and γ gives the total number of active warps. The objective of accessing the data from the global memory efficiently is to $Minimize(\sum_{i=1}^{\gamma} T_{iw})$, which is equivalent to $Maximize(\sum_{i=1}^{\gamma} Part_i)$.

In Fig. 2.10, partition camping effect is illustrated with an example. Let the streaming multi-processors in the Nvidia system be denoted by SM_i, where $1 \leq i \leq 30$. The active warps are given by W_i, where i is the streaming multiprocessor on which it is being executed. Now, if the data in the global memory is distributed in such a manner, that for a given instance of execution, all the active warps access data from the same partition, in this case $Partition_1$, then partition camping takes place. The data access is sequentialized for each of the warps, and all the other partitions

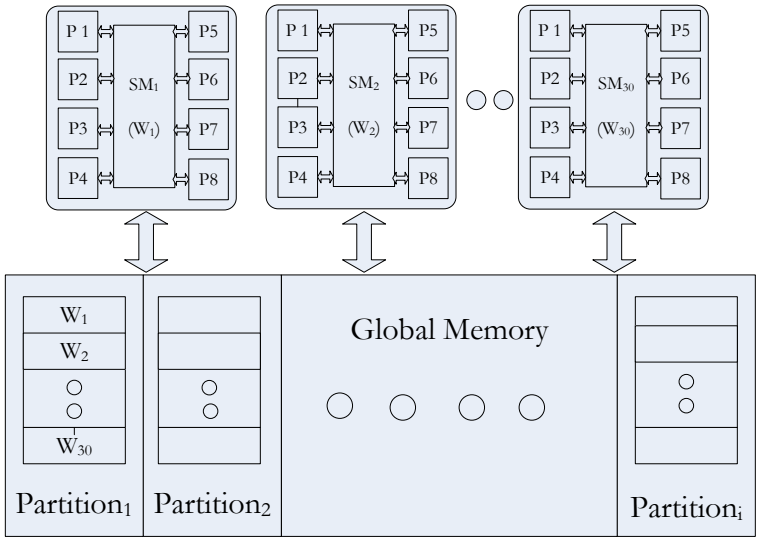


Figure 2.10: Partition Camping in Effect

are not accessed. The active warps accessing a particular partition are shown in a table inside the partition.

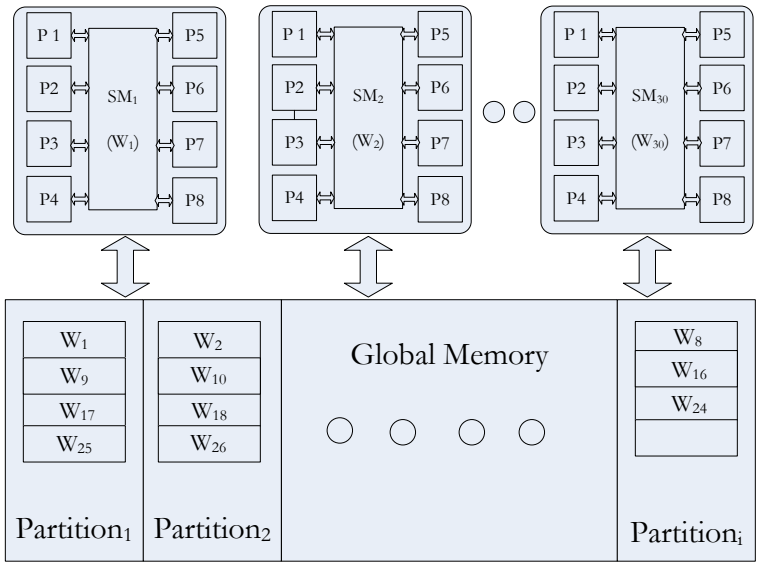


Figure 2.11: Avoiding Partition Camping by Distribution of Warps

In Fig. 2.11, a case where partition camping is avoided is shown. In this case, all the active warps are distributed evenly amongst the available partitions and the mapping is given by the following equation

$$Partition_{i\%p} \Leftarrow W_i \tag{2.3}$$

where, p is the total number of partitions available.

2.5 Summary

In this chapter we discuss about GPU architecture and CUDA. The CUDA programming model and memory model are studied and techniques to improve the performance of the same are discussed. Many different devices exist that are CUDA enabled and there have been changes with the availability of newer versions of the architecture. However, the basic principles of the models remain the same, and the discussions are valid across all the different architectures.

Chapter 3

Storing graphs on GPUs

3.1 Introduction

For performing computations on the GPUs storing the graph data on the device is essential. Using efficient data structures to store the data helps in making use of the level of memory with the least access latency. In this chapter we study the different methods to store graphs on the GPUs.

3.2 Related work

Using efficient data structures to store graphs for computation on both CPUs and GPUs have been studied extensively. Various modifications of the adjacency matrix and adjacency list data structures are considered.

Katz and Kider (Katz & Kider, 2008) and Buluc et al. (Buluç et al., 2010) proposed storing graphs on the GPU by dividing the adjacency matrix into smaller blocks. The required blocks are loaded in the memory, and after computation, are replaced by the next set of blocks. This representation still uses the adjacency matrix, and might include data which is not required for computations based on locality information.

Frishman and Tal (Frishman & Tal, 2007) propose representing multi-level graphs using 2D arrays of textures. They propose partitioning the graph in a balanced way, by identifying geometrically close nodes and putting them in the same partition. Since the partitions are based on locality information and balanced, it makes use

of the GPUs data parallel architecture. But partitioning the graph itself is a hard problem.

For sparse matrices, the Compressed Sparse Row (CSR) representation is useful (Garland, 2008). Also, representing the edge information using arrays to store the out-vertex and in-vertex numbers saves space for sparse matrices (Itokawa et al., 2007). This method is better suited for directed graphs.

Bader and Madduri (Bader & Madduri, 2008) proposed a technique where different representations are used depending on the degree of the vertices. This is relevant for storing graphs that exhibit the small-world network property, where vertices have an unbalanced degree distribution, with majority of vertices having small degrees and a few vertices are of very high degree.

Harish and Narayanan (Harish & Narayanan, 2007) describe the use of a compact adjacency list, where instead of using several lists, the data is stored in a single list. Using pointers for each of the vertices' adjacency information, data for the entire graph is kept in a single one dimensional array, which can be significantly large to be stored in the shared memory, and has been implemented in the global memory in their paper.

In this chapter, in addition to using the breadth-first search (BFS) information to carefully split the graph for processing, we introduce data structures for storing nodes and their adjacency information that are in contiguous levels of the BFS-tree. We propose both simple and modified data structures using least number of bits in addition to exploiting the symmetric property of undirected graphs. The modified data structure is similar to the one proposed by Harish and Narayanan (Harish & Narayanan, 2007), but with improvements, including the use of fewer bits in the general case and also using more than one array to store the entire adjacency data for the graph, thereby adhering to stricter memory requirement constraints.

3.3 Simple data structures for storing the graph information

Various data structures can be chosen to store the adjacency information of graphs in the GPU memory. Each of the different data structures have different storage requirements. There are several operations on graphs that need to access the stored data using one of the available data structures. A memory efficient data structure might have worse time complexity when it comes to accessing the required data for computations. Hence, there exists a trade-off between memory requirement and access time complexity, when choosing the appropriate data structure for storing the adjacency information of the graphs. In this section, we analyze the space required by different data structures and also the time complexity for performing common operations on graphs using the same. It must be noted that throughout the chapter, for calculations we use boolean data and it consists of a single bit, and does not refer to the data type available in programming languages.

3.3.1 Adjacency Matrix

For a graph $G = (V, E)$ with $|V| = n$, the size of adjacency matrix is n^2 bits, where each edge is stored using a single bit. To fit the adjacency matrix in the shared memory, the space required must be less than or equal to that of the desired level in the memory hierarchy of the GPU. Considering the architecture available on the Nvidia 10-series GPUs, for example C1060, the size of the shared memory is 16 KB. Hence, to satisfy the above constraint, $n^2 \leq 131,072$ (16 KB $= 16 \times 1024 \times 8$ bits $= 131,072$ bits), which gives $n \approx 360$. Therefore, using the adjacency matrix representation, the size of the largest graph that can be kept in the shared memory is 360 (assuming all shared memories in the different streaming multiprocessors contain identical data.)

3.3.2 Upper Triangular Matrix

The adjacency matrix representation contains redundant information, and those can be eliminated to reduce the storage requirements. For undirected graphs, values (i, j) and (j, i) are identical. So, storing only the Upper Triangular Matrix (UTM) of the adjacency matrix is enough, which requires $\frac{n\times(n+1)}{2}$ bits. So, the largest graph that can be kept in the shared memory using the UTM representation is 511. As all the values of $(i, i) = 0$, using the Strictly UTM representation (S-UTM) (i.e. without the data on the diagonal), size of the largest graph that can be kept in the shared memory is 512. Although the shared memory spans across 16 banks, it is preferable to store data for any specific node within a single bank thereby avoiding potential memory contention and reduce overall execution time. With the above requirement, the number of nodes that can fit in the shared memory is reduced to 506, where data is kept in increasing order of the node numbers. Also, using UTM representation, the number of nodes' data in each of the bank (size 8192 bits) varies, as shown in Table 3.1.

In the previous approach due to unbalanced distribution of nodes, threads assigned to operate on banks with more nodes would have to do significantly more work than the threads accessing banks with less number of nodes. On the other hand, if threads access a constant number of nodes it will result in inefficient memory utilization, thus limiting the overall size of graph that can be stored.

For load balancing the distribution can be done as follows. Using S-UTM representation different rows have different amount of data (see Table 3.2). To make the structure rectangular (see Table 3.3), the space gained by not storing redundant information in any row in the upper part of the S-UTM can be filled up by a corresponding row from the lower part. When n is even, the space gained in row i is filled with data values from row $n - i$ (see Table 3.3); when n is odd, the corresponding

Bank #	**Nodes in the Bank**	**# of Nodes**	**Space Required in bits**
0	0-15	16	7976
1	16-31	16	7720
2	32-48	17	7922
3	49-66	18	8073
4	67-85	19	8170
5	86-104	19	7809
6	105-124	20	7830
7	125-146	22	8151
8	147-169	23	8004
9	170-194	25	8100
10	195-221	27	8046
11	222-251	30	8085
12	252-285	34	8075
13	286-325	40	8020
14	326-378	53	8162
15	379-505	127	8128

Table 3.1: Distribution of nodes in the banks

space in row i is filled with data from row $n-(i+1)$ (see Table 3.5). In general, for any value of n the number of rows of data is reduced from n to $\lfloor \frac{n}{2} \rfloor$. This is called *Balanced* S-UTM (B-S-UTM), where all rows have the same amount of data, each corresponding to that of 2 nodes. The above method is similar to "rectangular full packed" in dense linear algebra (Gustavson et al., 2010). With the desire to store an entire row of data in a single streaming multiprocessor, this scheme also ensures all banks have equal number of nodes in them thereby achieving load-balancing. Using this scheme, the maximum number of rows that can be kept in a single bank is $\frac{(1024\times 8)}{511} \approx 16$. As there are 2 nodes' data in each row, the total number of nodes' data in each of the banks is 32. Therefore, the total number of nodes that can be kept in

this manner in the 16 banks is $32 \times 16 = 512$.

–	a	b	c	d	e	f	g
–	–	h	i	j	k	l	m
–	–	–	n	o	p	q	r
–	–	–	–	s	t	u	v
–	–	–	–	–	w	x	y
–	–	–	–	–	–	z	ϕ
–	–	–	–	–	–	–	ψ
–	–	–	–	–	–	–	–

Table 3.2: S-UTM for even number of nodes

a	b	c	d	e	f	g
h	i	j	k	l	m	ψ
n	o	p	q	r	z	ϕ
s	t	u	v	w	x	y

Table 3.3: S-UTM with load balanced approach for even number of nodes

–	a	b	c	d	e	f
–	–	g	h	i	j	k
–	–	–	l	m	n	o
–	–	–	–	p	q	r
–	–	–	–	–	s	t
–	–	–	–	–	–	u
–	–	–	–	–	–	–

Table 3.4: S-UTM for odd number of nodes

a	b	c	d	e	f	u
g	h	i	j	k	s	t
l	m	n	o	p	q	r

Table 3.5: S-UTM with load balanced approach for odd number of nodes

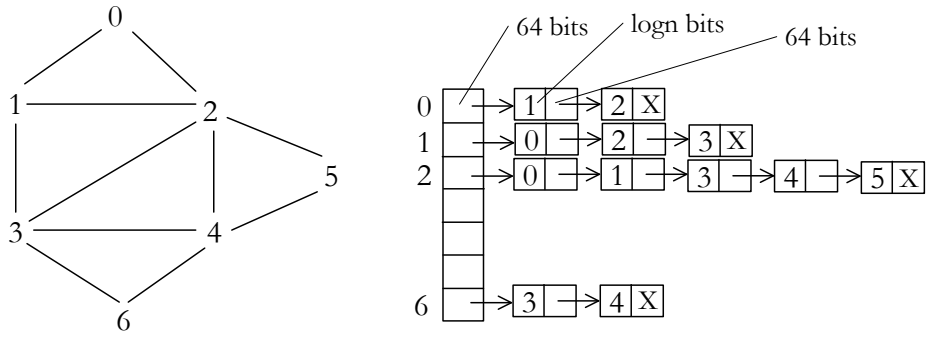
The number of simultaneous thread executions is limited by the number of GPU processors in a streaming multiprocessor. Each thread is allocated a set of combinations of nodes (with cardinality k) and a thread is to determine if the desired property (e.g., do they form a connected subgraph?) holds. In order to use all available GPU processors in other streaming multiprocessors, we have to duplicate the graph and place it on all the shared memories on other streaming multiprocessors. Care must be taken to ensure each thread is given a unique set of combinations to test, to avoid duplication in work.

The sets of combination of k nodes are allocated to each streaming multiprocessor as follows. Since the shared memory in each streaming multiprocessor can store up to 512 nodes, it can be assumed that there are 512 sets of combinations, each starting with a unique node number. We allow the first 29 streaming multiprocessors to operate on 17 unique sets of combinations each and the last streaming multiprocessor operates on the remaining 19 sets ($17 \times 29 + 19 = 512$). In each streaming multiprocessor, depending on the number of threads the unique sets are uniformly divided to be processed.

3.3.3 Adjacency List Using Array of Linked Lists

Other than the adjacency matrix, adjacency list is also a common data structure used to store graph information. There can be various modifications in the implementation of the adjacency list, and each has different memory requirements and data access complexity. In this sub-section, we study the implementation of the adjacency list using an array of linked lists, also referred to in here as AL-AL, and is shown in Fig. 3.1.

Let the total number of nodes in the graph be n and total number of edges be m. Here, all the nodes are stored in an array; the identifiers of the nodes given by the indices of the array location. Each array element contains a pointer to the starting



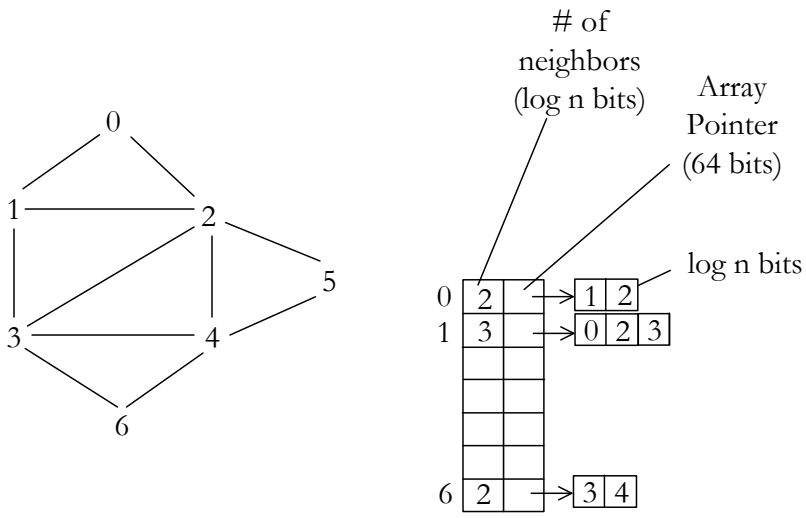
An example graph (left) with the corresponding adjacency list representation (right) using Array of Linked Lists (AL-AL)

Figure 3.1: Adjacency List Using Array of Linked Lists

node of the linked list representing the neighbors of the corresponding node i.e., the edge information. In the example graph shown in Fig. 3.1, node 0 is connected to nodes 1 and 2. Therefore, a pointer to node 1 is stored in the array index corresponding to node 0. The storage for node 1 contains the identifier for the node, and a pointer to the next neighbor i.e., node 2. Since, there are no more neighbors for node 0, the pointer associated with node 2 contains an invalid marker, denoted in Fig. 3.1 by "X". Now, each of the edges would be stored two times, once for each end vertex. Considering a 64-bit machine, each of the pointers require 64 bits. So, the space needed to store the array containing the nodes is given by $n \times 64$ bits. Now, each of the edges are represented by the end vertex number, and also contains a pointer to the next edge information if there exists another edge for the node under consideration. Since there are n nodes in the graph, representing a node number requires $\log n$ bits. Hence, for storing each of the edges, the space required is $(\log n + 64)$ bits, giving a total of $2m \times (\log n + 64)$ bits for all the edges. Therefore, the total size required for this representation is given by $n \times 64 + 2m \times (\log n + 64)$ bits.

3.3.4 Adjacency List Using Array of Arrays

In the adjacency list representation using array of linked lists, there are too many pointers involved in the implementation, thereby increasing the size of the data structure. Using arrays to group data together replacing pointers can reduce the memory required, and this is referred to as adjacency list using array of arrays, and is shown in Fig. 3.2.



An example graph (left) with the corresponding adjacency list representation (right) using Array of Arrays (AL-AA)

Figure 3.2: Adjacency List Using Array of Arrays

Here the edges from each node are stored using arrays instead of linked list. Since there are no pointers involved between edges, to determine the size of the array, the number of neighbors for each of the nodes need to be stored. In this case, all the nodes in the graph are stored in an array, along with the number of neighbors and in addition there is a pointer for each of the nodes to the array containing its neighbors. In the example graph shown in Fig. 3.2, node 1 is connected to 3 nodes, numbered 0, 2 and 3. Therefore, for node 1, a value of 3 is stored corresponding to the number of neighbors, and also a pointer to the array of its neighbors, which contains

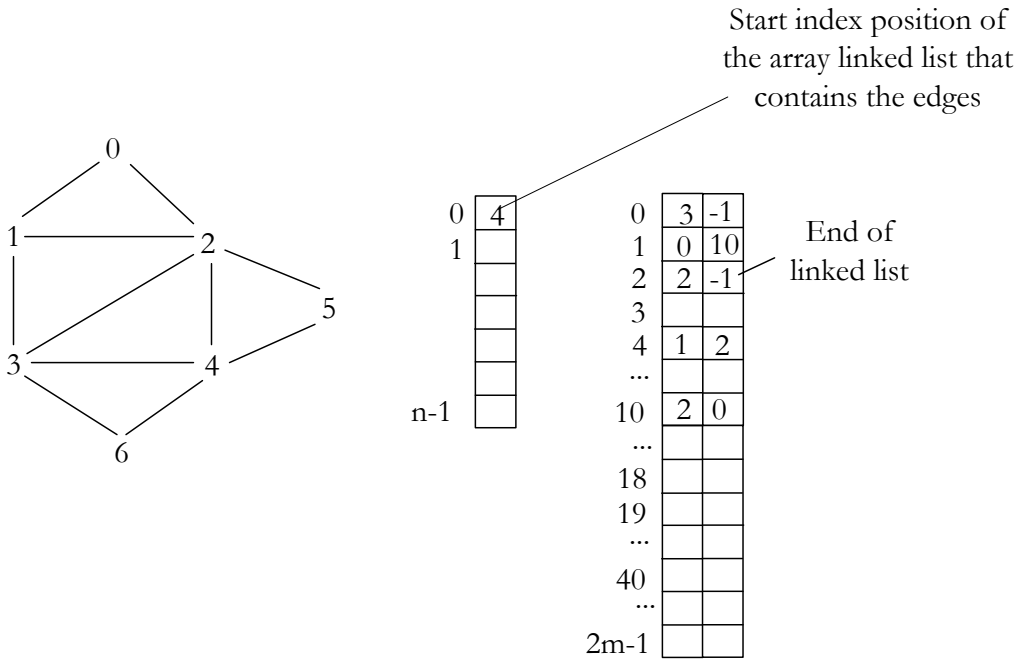
the identifiers for the nodes 0, 2 and 3. The size of the array with the number of neighbors and pointers for the n nodes require $n \times (\log n + 64)$ bits. Additionally, the total space required for all the arrays containing information about the m edges is $2m \times \log n$ bits. Therefore, the total size required for this representation in bits is given by $n \times (\log n + 64) + 2m \times \log n$.

An improved version of this data structure that requires less storage is one which does not store the number of neighbors for each of the nodes. However, this information is required to retrieve the adjacency data, and must be available during computation. This can be achieved by using the sizeof() function on the array containing the neighbors to find the required number at runtime. Hence, the array of arrays variant with the sizeof() function requires $n \times 64 + 2m \times \log n$ bits.

3.3.5 Adjacency List Using Array Implementation of Linked Lists

In the adjacency list representation using array of arrays, there are still pointers involved, and that requires significant amount of space. Instead of using pointers, the information for the edges can be stored using an array and relevant identifiers. So, the linked list pointers are replaced by using arrays. This representation is referred to as the adjacency list using array implementation of linked lists, and is shown in Fig. 3.3.

Here, both the nodes and edges are stored in arrays. Each element for the node array contains the starting index position in the array containing the edges. Since there are $2m$ indices in the edge array, the values stored in the node array each require $\log 2m$ bits, giving a size of $n \times \log 2m$ bits for the node array. The edges are represented by a pair of elements in the edge array. The first element contains the end vertex number of the edge, and the second element contains the index position of the next edge information in the array, or an identifier value of "-1" which indicates the end of the linked list. As there are n nodes, the first element requires $\log n$ bits,



An example graph (left) with the corresponding adjacency list representation (right) using Array Implementation of Linked List (AL-ALL)

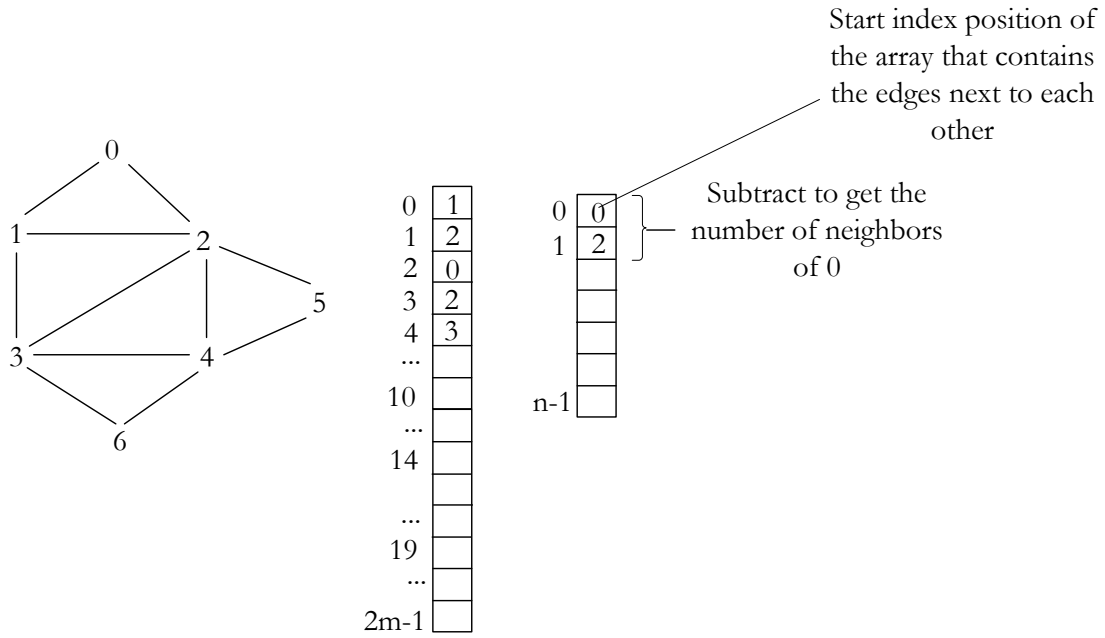
Figure 3.3: Adjacency List Using Array Implementation of Linked Lists

and for the $2m$ indices possible, the second element requires $\log 2m$ bits. So, the size of the edge array is $2m \times (\log n + \log 2m)$ bits. Therefore, the total size required for this representation in bits is given by $n \times \log 2m + 2m \times (\log n + \log 2m)$.

Now, the data representing the edges can be stored in any order in the array implementing the linked list. In the example graph shown in Fig. 3.3, the neighbors of node 0 are stored starting at position 4 in the array just to illustrate the fact that data can be stored at any location in the array as the pointers would correctly determine the location of the next available data. Node 1, which is a neighbor of node 0 is stored at location 4, and contains a pointer to location 2. Location 2 contains the identifier of the other neighbor of node 0 i.e., node 2 and contains an identifier "-1" for the pointer indicating the end of the list since there are no more neighbors

for node 0.

3.3.6 Adjacency List with Edges Grouped



An example graph (left) with the corresponding adjacency list representation (right) with edges grouped (AL-EG)

Figure 3.4: Adjacency List with Edges Grouped

In the array implementation of linked lists, the edges are stored in an array using arbitrary indices which requires storing the next index position for each of the edges. This can be improved by storing all the edges next to each other. In this manner, neither the total number of edges for each of the node nor the next index positions are required. As before, both the nodes and edges are stored in arrays. The node array elements point to the starting index position of the edges in the edge array, and all the edges are stored in consecutive locations. The total number of edges for each of the nodes can be easily calculated by subtracting the starting index position of the current node with that of the next node. This representation, similar to the

one described in (Harish & Narayanan, 2007) as compact adjacency list, is referred to as the adjacency list with edges grouped, and is shown in Fig. 3.4.

In the example graph shown in Fig. 3.4, node 0 has two neighbors, nodes 1 and 2. This information is stored in the edge array using consecutive locations starting from 0. Therefore, the node array element corresponding to node 0 contains the value of location 0. For node 1, the neighbors 0, 2 and 3 are stored in the edge array starting at location 2, and this value is stored in the node array. Using the information in the node array for nodes 0 and 1, the number of neighbors for node 0 can be found by subtracting the corresponding value of node 0 from that of node 1.

As in the case of array implementation of linked lists, the space required for the node array is $n \times \log 2m$ bits. For the edge array, there are $2m$ elements, and each require $\log n$ bits. So, the size of the edge array is $2m \times \log n$ bits. Therefore, the total size required for this representation in bits is given by $n \times \log 2m + 2m \times \log n$.

3.4 Operations on graphs

For solving various problems on graphs, different operations need to be performed on the graph data. In this sub-section we consider two types of operations on graphs - a) Query operations, and b) Update operations. The different types of query operations can be the following:

- $N(v)$: List the neighbors of node v.
- $D(v)$: Find the number of neighbors of node v.
- $Edge(u, v)$: Determine if there is an edge between vertices u and v.

The different types of update operations can be listed as follows:

- $Add(v)$: Add a vertex v.

- *Add*(u, v): Add an edge between the vertices u and v.
- *Remove*(v): Remove the vertex v and all corresponding edges (u, v).
- *Remove*(u, v): Remove the edge (u, v).

As mentioned earlier, there is a trade-off between the space required and access time complexity for different data structures. The various query and update operations provide a measure for the time complexity. For a given graph G with degree $\delta(G)$ consisting of n nodes and m edges, Table 3.6 provides a summary of the comparison of the time complexity for the operations on graphs using the data structures discussed in this section.

3.5 Advanced data structures

In order to process the reduced combinations described in the previous subsection, streaming multiprocessors should be responsible for generating only relevant combinations, by knowing the number of nodes in each level and BFS numbering of the nodes. The entire graph can be stored using any of the efficient data structures discussed in the previous section. For the purpose of discussion, let the graph be stored using S-UTM representation along with the information of the number of nodes at each level. Instead, it may be beneficial in some cases to store adjacent levels of the BFS-tree. Thus, except for the node in the root level for each set of adjacent levels, there will be a S-UTM data structure along with the starting node number and the total number of nodes. This representation is called S-UTM-ADJ. Similarly, such corresponding representations exists for the other data structures too.

In addition to the S-UTM-ADJ we have devised another data structure called *Parent Array Representation* (PAR) wherein each node keeps information of its parent along with adjacent nodes in both the parent's and the same level as a list. By keeping additional information we can further reduce the space requirements as shown below.

Data Structure	***N(v)***	***D(v)***	***Edge (u, v)***	***Add(v)***	***Add (u, v)***	***Rm. (v)***	***Rm. (u, v)***
Adjacency matrix	n	n	1	n^2	1	n^2	1
S-UTM	n	n	1	n^2	1	n^2	1
Array of linked lists	$\delta(G)$	$\delta(G)$	$\delta(G)$	n	1	$n+m$	$\delta(G)$
Array of arrays	$\delta(G)$	1	$\log \delta(G)$	n	$\delta(G)$	$n+m$	$\delta(G)$
Array of arrays with sizeof()	$\delta(G)$	1	$\log \delta(G)$	n	$\delta(G)$	$n+m$	$\delta(G)$
Array implementation of linked lists	$\delta(G)$	$\delta(G)$	$\delta(G)$	n	1	$n+m$	$\delta(G)$
Adjacency list with edges grouped	$\delta(G)$	1	$\log \delta(G)$	n	m	$n+m$	m

Table 3.6: Comparison of time complexity for operations on graphs using different data structures

3.5.1 Parent Array Representation:

In PAR, for each of the nodes contained in k-levels, the following information is stored: a) The parent node number, b) An identifier (0 or 1) to specify if there are other neighboring nodes belonging to the same level (siblings) or previous level (i.e., parent's level), c) If the identifier value is 1, then the number of neighbors identified in the previous step, d) Node numbers for each of the neighbors to identify them. Fig. 3.5 shows an example of PAR for an arbitrary graph. The node numbers are not stored explicitly but calculated from a value x, which gives the starting node number.

As the node numbers are in strictly increasing order, the calculation is simple. Also, neighboring node numbers are not stored explicitly, but calculated from another value y, which is the parent number for x.

The space required is therefore reduced by storing only the differences in the numbering of the parents and neighbors for the nodes, which depends on the value δ (degree of the graph.) While storing the differences between the numbering, the worst case graph, shown in Fig. 3.6, would give the difference in the order of n. For example, if a level has p nodes, there would be p parent nodes, and say there are other q neighboring nodes. If the node numbers are stored, it would take $((2 \times p + q + p') \times \log_2 n + p)$ bits, where the extra p bits are the identifiers. Also $p' \leq p$ are nodes that have other neighbors and need an additional value indicating the total number of such nodes. Whereas, if the differences in the numbering is used, then it would take in the worst case, $((p + q) \times \log_2 \delta + (2 + p') \times \log_2 n + p)$ bits.

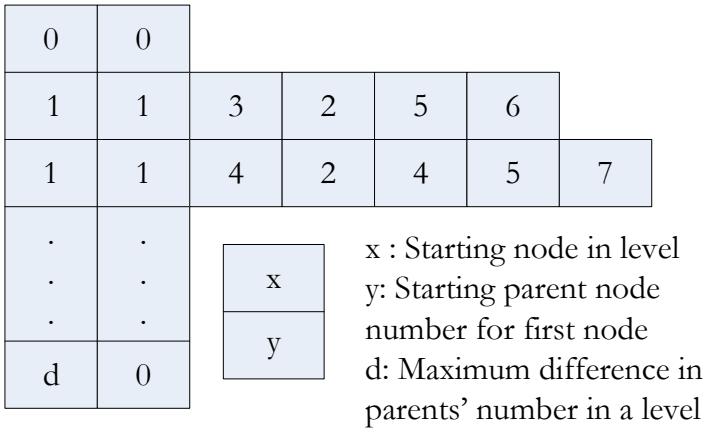


Figure 3.5: PAR for an arbitrary graph

Interestingly, more than one type of storage mechanism can be used depending upon the structure of the BFS-tree. Each of the levels of the BFS-tree would use either *UTM* or *PAR*. The graph is preprocessed in the CPU, and depending on the size of the representations, the smaller data structure is chosen.

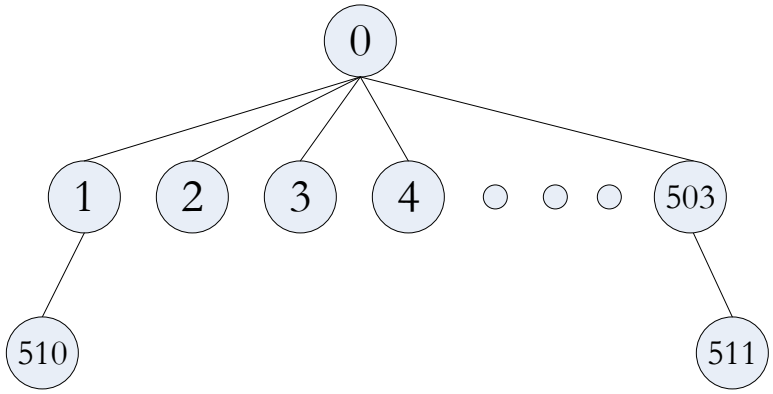
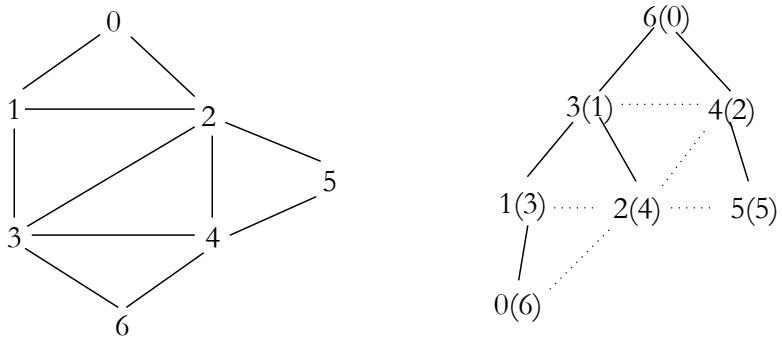


Figure 3.6: Worst Case: d is order of n

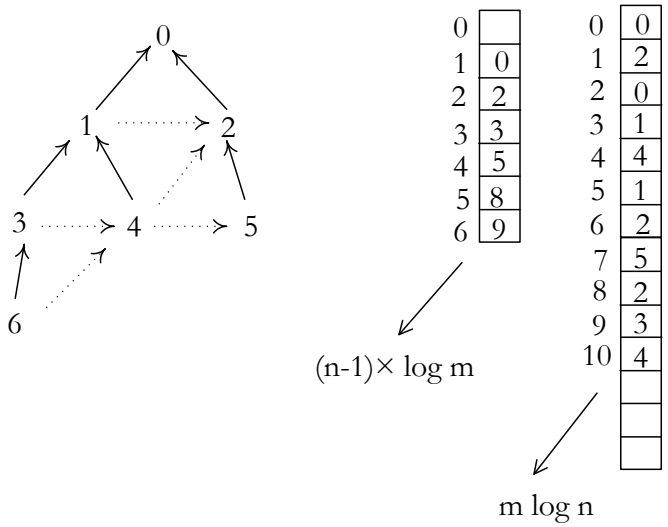
3.5.2 Directed BFS-tree and AL-EG Storage

Another data structure that uses breadth-first search tree information and is proposed in this chapter is referred to as the directed BFS-tree with adjacency list and edges grouped storage. Here, first the breadth-first search tree is constructed and nodes are provided with BFS numbers. The following links are considered as directed links: a) Child to parent in the BFS tree, b) all non-tree edges from node at level l to nodes at level $l-1$ and, c) all non-tree edges from nodes numbered i to nodes numbered j such that $i < j$ and nodes i and j are at the same level in the BFS tree. This representation is shown with an example in Fig. 3.7.

For the example graph in Fig. 3.7, the breadth-first search is performed starting with node number 6. The BFS-tree is shown with the original node numbers along with the BFS numbering in parentheses. Now, only the directed links as defined above are stored in the data structure. So, for the renumbered node 0, there is no information to be stored in the edge array. For the renumbered node 1, there are directed links to nodes 0 and 2, and these are stored in the edge array starting at location 0. Similarly, for node 2, necessary information is stored in the edge array starting at location 2.



An example graph (left) with the corresponding BFS-tree renumbering (right)



Directed-BFS tree (left) with the corresponding adjacency list representation (right) with edges grouped (AL-EG)

Figure 3.7: Directed-BFS tree with Edges Grouped

The combinations of nodes to be tested for the counting problems are generated in increasing order of node numbers. So, the above information is sufficient to perform the various required operations on graph data. The main advantage of this data structure over the adjacency list with edges grouped is the data for the edges are stored once instead of twice. For this data structure, the total memory requirement

is given by $(n-1) \times \log m + m \times \log n$ bits.

3.6 Comparison of different data structures

In the above sub-sections, different data structures that can be used to represent the adjacency information for graphs are discussed. Gradual modifications are made to reduce the space requirements by eliminating the need for pointers and rearranging the data. For a graph with n nodes and m edges, space requirements for the different data structures can be summarized as given in Table 3.7.

Data Structure	**Total bits**
Adjacency matrix	n^2
S-UTM	$n \times (n-1)/2$
Array of linked lists	$n \times 64 + 2m \times (\log n + 64)$
Array of arrays	$n \times (\log n + 64) + 2m \times \log n$
Array of arrays with sizeof()	$n \times 64 + 2m \times \log n$
Array implementation of linked lists	$n \times \log 2m + 2m \times (\log n + \log 2m)$
Adjacency list with edges grouped	$n \times \log 2m + 2m \times \log n$

Table 3.7: Comparison of memory requirements for the different data structures

An example graph of size 16 is provided for illustration. The structure of the original graph and a BFS-tree is shown in Fig. 3.8. Table 3.8 compares space requirements for the different data structures.

In case of the S-UTM-ADJ, PAR and the Hybrid data structures, the graph data is split and stored according to level information. For the rest of the data structures, the adjacency information of the entire graph is kept together. Although, from the data in Table 3.8, it seems S-UTM is the best choice when storing the entire graph together, for larger size sparse graphs other data structures might be more efficient. For example, if for a given graph $n = 300$ and $m = 900$, the corresponding storage requirements in bits for S-UTM is $44,850$; whereas that for the adjacency list with

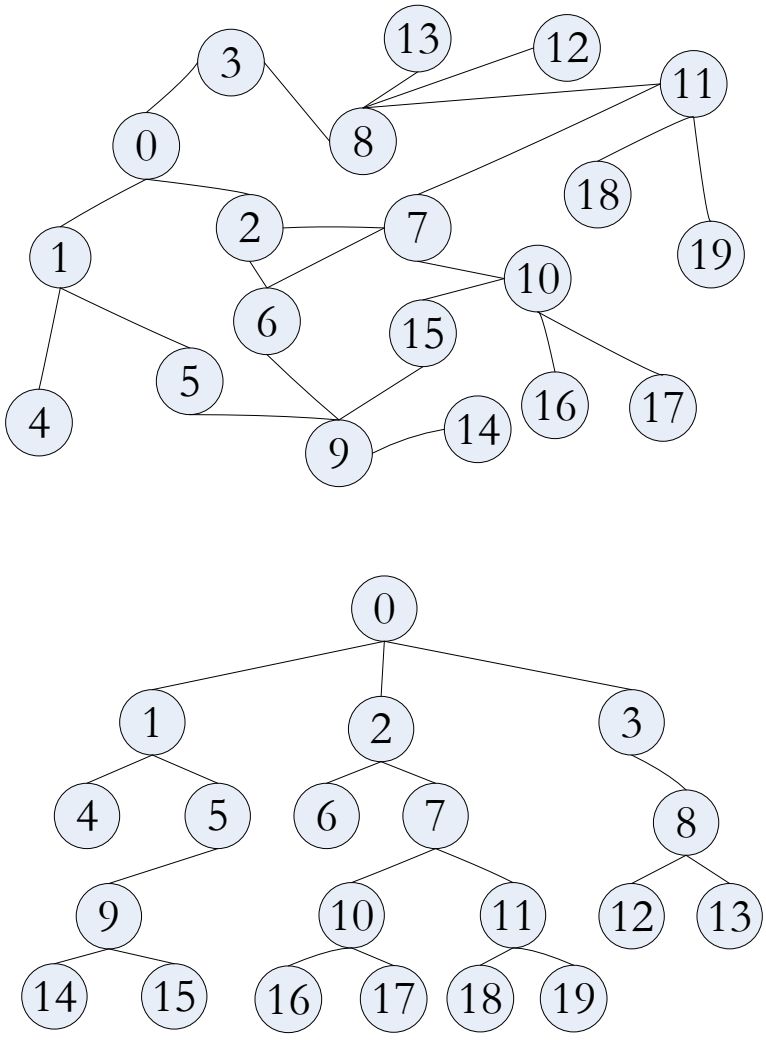


Figure 3.8: Sample graph (top) and BFS-tree for comparing data structures (bottom)

edges grouped is $18,057$ bits and for directed-BFS tree and AL-EG storage is $10,341$ bits.

3.7 Summary

Storing data on the GPU efficiently plays an important part in using the device for computations. In this chapter we study both naive and advanced data structures as well as propose novel techniques for storing graph data on the GPU. Starting from adjacency matrix, we move into more complex data structures that exploit the

Data Structure	**Space Required (bits)**
Array of linked lists	4287
Array of arrays	1557
Array of arrays with sizeof()	1471
Array implementation of linked lists	540
Adjacency matrix	400
Adjacency list with edges grouped	300
Upper Triangular Matrix	210
Strictly-UTM	190
Directed-BFS and AL-EG	180
S-UTM-ADJ	174
Parent Array Representation	113
S-UTM-ADJ/PAR: Hybrid	110

Table 3.8: Comparison of space requirements

structure of the data being considered too. Using various data structures not only have different size requirements but also have varying memory access complexities; and this aspect has also been analyzed here. Certain sections of this chapter are adapted from our research paper (Chatterjee et al., 2013a).

Chapter 4

Counting Problems on Graphs

4.1 Introduction

The availability and utility of large numbers of Graphical Processing Units (GPUs) have enabled parallel computations using extensive multi-threading. Sequential access to global memory and contention at the size-limited shared memory have been main impediments to fully exploiting potential performance in architectures having a massive number of GPUs. We propose novel memory storage and retrieval techniques that enable parallel graph computations to overcome the above issues. More specifically, given a graph $G = (V, E)$ and an integer $k <= |V|$, we provide both storage techniques and algorithms to count the number of: a) connected subgraphs of size k; b) k cliques; and c) k independent sets, all of which can be exponential in number. Our storage technique is based on creating a breadth-first search tree and storing it along with non-tree edges in a novel way. We also provide an algorithm for counting triangles in graphs.

The counting problems mentioned above have many uses, including the analysis of social networks. Counting the number of subgraphs matching given templates is used in data mining for advertising and fraud detection among others (Zhao et al., 2012). The counting problems are also relevant for studying molecular networks and finding molecules with specific substructures. Finding "motifs" i.e., frequent subgraphs in protein-protein interaction networks also benefit from the problems described above, and is useful in drug discovery by screening large number of molecules, and also

chemical synthesis success prediction (Borgelt & Berthold, 2002).

4.2 Related work

Different data structures and modifications, combined with various optimization techniques are considered for efficiently solving problems on graphs. The algorithm described in (Katz & Kider, 2008) by Katz and Kider handles graph sizes that are inherently larger than the DRAM memory available on the GPU by dividing into smaller blocks. The shared memory cache efficient algorithm to solve transitive closure and all-pairs shortest path problem on the GPU in (Katz & Kider, 2008) deals with directed graphs for large data sets. However, it does not explicitly propose techniques to use the global memory which would be essential to store such data.

Bordino et al. (Bordino et al., 2008) have developed a technique for counting subgraphs of size 3 and 4 for large streaming graphs. The counting algorithm is sequential in nature and can be used in many applications.

Vineet et al. (Vineet et al., 2009) study an algorithm for fast minimum spanning tree for large graphs on the GPU. The authors in (Vineet et al., 2009) focus on the algorithmic aspect, rather than overcoming the hardware limitations encountered while using the global memory as the storage.

Buluc et al. (Buluç et al., 2010) look at an alternative way of storing the graphs on the GPU by dividing the adjacency matrix into smaller blocks. These representations, although efficient in terms of size of the data structure, might not be ideal for data stored on the global memory, to be operated by a large number of threads as available on the GPUs, causing memory accesses to become sequential. Data structures with redundant information might be more suited to efficiently utilize the larger global memory by avoiding sequential data accesses.

Harish and Narayanan (Harish & Narayanan, 2007) describe methods to accelerate

large graph algorithms on the GPU using CUDA. Although the algorithms are implemented in the global memory using compacted adjacency list, there is no discussion on using available primitives to offset the usage of the slower memory.

Hong et al. (Hong et al., 2011) discuss techniques to improve the work balance among active threads in the level of warps for maximizing the efficiency for solving graph algorithms on CUDA. The methods suggested in (Hong et al., 2011) study the trade-off between achieving load balancing and utilization of the GPU cores. But, there is a lot of fine-tuning involved and the parameters for the optimum case are dependent on the structure of the input graph.

Ruetsch and Micikevicius (Ruetsch & Micikevicius, 2009) have proposed techniques to use available primitives in the context of matrix transpose problem. The results show that tuning the data structures depending on the application can yield improved bandwidth for the data access from the global memory which is comparable to that of the shared memory. Our partition camping avoidance techniques are based on similar principles. However, the data structures used in our research and the computations are entirely different based on the additional information that we incorporate by studying the breadth-first search tree properties of the input graph.

Yang et al. (Yang et al., 2010b) provide general methods for constructing an optimization compiler for GPUs. In (Yang et al., 2010b), the authors study and analyze the basics of identifying the scope for efficient data access from the global memory in the context of matrix multiplication. The main focus in (Yang et al., 2010b) is on the automatic transfer of the code by the optimizing compiler using various other methodologies, including the usage of shared memory and re-organizing the thread blocks for efficient code generation.

Counting triangles have many applications including the analysis of social networks as described in (Tsourakakis et al., 2009) (Becchetti et al., 2008). Simple data structures, including those storing redundant information are considered to take ad-

vantage of the available memory access optimizers like using memory coalescing and avoiding partition camping.

4.3 Counting connected subgraphs

Given a graph $G = (V, E)$ with $|V| = n$ and a value k as input, the problem is to count the number of connected subgraphs of size k in G. This can be done using a brute-force approach. The procedure is as follows. The nodes of the graph are numbered from 0 to $n - 1$. All possible combinations of size k are generated one by one from n nodes. For each of the generated combinations, a breadth-first search (BFS) is performed, and if all the nodes are reachable starting from the first node, then the subgraph is connected, otherwise it is not. For a complete graph, the total number of such connected subgraphs is ${}^nC_k$, and this is the number of combinations that is required to be tested for any graph. The node numbers of the subgraphs are not stored for space constraints; only the total count for the connected subgraphs is reported. However, in this approach the total number of combinations to be checked is huge, and this can be effectively reduced as discussed in the following Section.

4.3.1 Using BFS-tree information

Considering a breadth-first search (BFS) representation of the graph, nodes chosen in any combination must be in k adjacent levels, otherwise the subgraph containing the nodes in the combination will not be connected. For example, let $k = 3$ and a combination contain the nodes 10, 12, and 14 at levels 4, 5, and 7, respectively in the BFS-tree. It is possible for nodes 10 and 12 to be connected by an edge since they are in adjacent levels. For the sake of discussion, assume there is an edge (10, 12) in the graph. It follows that there cannot be a edge (10, 14), for otherwise the level of node 14 in the BFS-tree must be 3, 4, or 5. A similar argument can be made for the

edge (12, 14) and hence the graph induced by those vertices is not connected. From the above reasoning it is evident that using the graph's BFS-tree can be an effective tool in reducing the number of combinations to be tested. We will further examine this in the following subsections.

4.3.2 BFS-tree node numbering

The nodes in the BFS-tree are numbered in the order they are visited following a breadth-first search of the graph. Any arbitrary node is chosen as the starting or root node, and is numbered 0 and belongs to the first level. All nodes that are neighbors of the first node belong to the second level. Similarly, any unvisited node that is neighbor of the nodes in the *previous* level, belong to the *next* level.

4.3.3 BFS-tree properties and applications

The following are some of the properties of BFS-tree that are useful in the study of graphs.

1. If two nodes are neighbors in the original graph, their level numbers cannot differ by more than one. Let v_i and v_j be adjacent nodes belonging to levels l_{v_i} and l_{v_j} respectively in the BFS-tree. So, from this property we have $|l_{v_i} - l_{v_j}| \leq 1$.

2. Any node in a level with a parent numbered α can also be neighbors with nodes numbered greater than α in the level of α, but not less. Let node v_i be the parent of node v_j and Δ_j be the set of neighbors of v_j. Therefore, for any node v_t where $l_{v_i} = l_{v_t}$, $v_t \notin \Delta_j \forall t < i$.

3. The structure and *height* of the BFS-tree depends on the choice of the starting or root node.

4.3.4 Reducing number of combinations to be tested

In the case of purely random distribution of nodes all possible combinations must be tested. So, for n nodes and subgraphs of size k, the total number of combinations to be tested is ${}^nC_k$. If $n = 360$ and $k = 10$, for example, the corresponding value is ${}^{360}C_{10} \approx 8.88 \times 10^{18}$.

In the case where a BFS-tree of the graph fits in the shared memory, using its properties the number of combinations to be tested can be drastically reduced. The idea here is to test for combinations with nodes in each of the consecutive k levels of the graph. Let $n = 360$ and $k = 10$, and the number of nodes in each level of the graph be 20. Then the total number of levels L in the graph is $\frac{360}{20} = 18$. Therefore, the number of consecutive k levels is $L - k + 1 = 9$. Now, for each of these set of levels, k nodes are to be chosen. The number of combinations to be tested taking different number of levels at a time is given as follows:

Considering 1-level i.e., for each of the different levels: ${}^{20}C_{10} = 184,756 \approx 1.84 \times 10^5$. Considering 2-levels: $({}^{20}C_1 \times {}^{20}C_9 + {}^{20}C_2 \times {}^{20}C_8 + {}^{20}C_3 \times {}^{20}C_7 + {}^{20}C_4 \times {}^{20}C_6) \times 2 + {}^{20}C_5 \times {}^{20}C_5 = 847,291,016 \approx 8.47 \times 10^8$. Considering for 3-levels: $72,851,600,250 \approx 7.28 \times 10^{10}$. Similarly, we can calculate the number of combinations taking 4 levels to 10 levels at a time.

Considering the general case, where there is a total of n nodes divided among L levels, such that each level consists of $\frac{n}{L}$ nodes say p. Then, the number of combinations to be checked for k node subgraph is: $\sum {}^pC_{n_1} \times {}^pC_{n_2} \times \cdots \times {}^pC_{n_k}$ such that $\forall n_i \geq 1, \exists n_1 + n_2 + \cdots + n_k = k$.

When taking 1-level at a time, the number of combinations to be tested is $\approx 1.84 \times 10^5$. But, all available L levels will be tested for these many combinations when they are considered individually. So, total number of combinations when considering all the 1-levels is $\approx 1.84 \times 10^5 \times L$. Similarly, taking 2-levels at a time, the number of

combinations to be tested is $\approx 7.28 \times 10^{10}$. But, there are $L-1$ such combinations of consecutive 2-levels. So, the total combinations when considering the contribution of all the 2-levels is $\approx 8.47 \times 10^8 \times (L-1)$. There are correspondingly $L-2$ combinations for consecutive 3-levels, $L-3$ combinations for 4-levels and so on. In general, there are $L-(k-1)$ such combinations for consecutive k levels. Therefore, the total number of combinations to be tested is given by: $1.84 \times 10^5 \times 18 + 8.47 \times 10^8 \times 17 + 7.28 \times 10^{10} \times 16 + 1.35 \times 10^{12} \times 15 + 9.82 \times 10^{12} \times 14 + 3.5 \times 10^{13} \times 13 + 6.9 \times 10^{13} \times 12 + 7.6 \times 10^{13} \times 11 + 4.3 \times 10^{13} \times 10 + 1.02 \times 10^{13} \times 9 = 2.8 \times 10^{15}$ i.e., about $2,800$ trillion. The number of combination testing decreases by $\approx 8.87 \times 10^{18}$.

4.3.5 Splitting for larger graphs

In the previous sections, we have seen if the graph fits in the shared memory, we can use one of the several techniques based on BFS-tree. In this section we show how to process graphs where the BFS-tree does not fit in the shared memory.

We make the following assumption: Given a graph G, there exists a BFS-tree T, such that the graph induced by nodes in any consecutive k levels of T has connected components of size less than 512 in G. Additionally, we assume the entire graph can fit in the shared memory of the 30 streaming multiprocessors. It must be noted that the value 512 corresponds to using S-UTM as the data structure for storing the graph; using other data structures would give different values. However, the underlying principles discussed in this section are applicable to splitting for larger graphs irrespective of the choice of data structures.

In the shared memory of each of the consecutive streaming multiprocessors i, nodes in level $k+i$ are added and nodes in level i removed. In this case, if the average number of nodes in each of the levels is l_{avg}, then the total number of nodes that can fit in the shared memory is $512 + l_{avg} \times 29$, where $l_{avg} \leq 512$. This is an improvement over the schemes presented in Section 3.3 wherein we could process at most 512 nodes

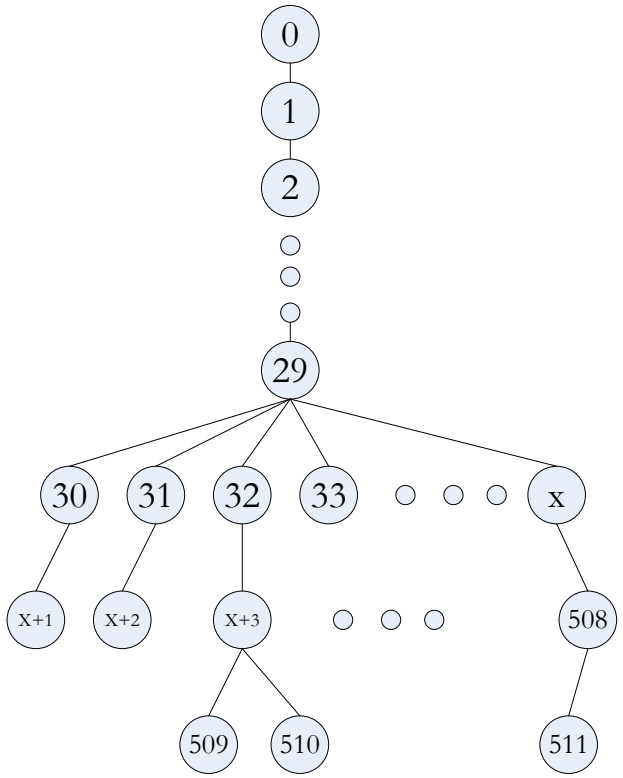


Figure 4.1: Worst Case BFS-tree for multiple streaming multiprocessors

even when using the shared memory on all the streaming multiprocessors. In the worst case, if the first 29 levels of T has single nodes, then only 1 new node can be brought in while storing the next consecutive k levels, giving a maximum size graph of $512 + 29 = 541$ nodes. An example BFS-tree of the worst case graph is shown in Fig. 4.1.

Now, let us assume there exists consecutive k levels of T that do not fit in the shared memory. In this case finding connected components in the graph induced by the nodes in the given k levels might be helpful. As nodes in separate connected components cannot be part of connected subgraphs, calculations can be done on them separately. If there are more than one connected components where each has less than 512 nodes, then calculations can proceed in the following manner; else we have to use the global memory to store them. If any of the components has less than k nodes,

then those can be excluded from the calculations. The other components can be kept in the shared memory of a streaming multiprocessor separately, or with other connected components provided the total number of nodes in all does not exceed 512. Now, all possible combinations of size k can be tested from among the nodes of each of the connected components separately.

Algorithm 1: Counting subgraphs of size k using all streaming multiprocessors by splitting G horizontally using T

```
Input: BFS-tree T of graph G
Output: Total count of connected subgraphs of size k
begin
    {L_k} ← divIntoKLevelSets(T);
    {L_k}, {M} ← resetLevlsModls({L_k}, {M});
    TotalCount ← 0;
    while there are sets of levels marked as new do
        if selected set is the last one then
            while levels available in the set do
                if there are k levels in the memory then
                    Clear memory, mark streaming multiprocessor available;
                else
                    curLvl ← nextAvailableLvl;
                    TotalCount ← TotalCount +
                    testCon(GenNxtComb((curLvl)));
                    while there are previous levels do
                        curLvl ← curLvl ∪ prevLvls;
                        TotalCount ← TotalCount
                          + testCon(GenNxtComb(curLvl));
                    end
                end
            end
        else
            TotalCount ← TotalCount + testCon(GenNxtComb(fstLvl));
            TotalCount ← TotalCount + testCon(GenNxtComb((allKLvls)));
        end
    end
end
```

Algorithm 1 takes T as input, and checks for connected subgraphs. The algorithm makes use of all available 30 streaming multiprocessors in the GPU by dividing the

work among them. The number of sets of k consecutive levels if there are L levels in T is $Q = L-k+1$. If $Q > 30$, the remaining sets are brought from the global memory after the current round of operations are completed. With this approach graphs that are stored even in external memories can be processed.

Overview of Algorithm 1: T is divided into sets of k consecutive levels and each is processed by a streaming multiprocessor. The beginning level in the set is processed first, and if it contains more than k nodes, then subgraphs of size k are checked for connectedness from among the nodes in that level. Then all the nodes in the k levels are considered together, and tested for combinations by the function *GenNxtComb(Nodes)* where each combination contains at least one node from the beginning level and one from the rest of the levels to avoid redundant checking. The above procedure is done for all but the last set of k levels. For the last set, each new level is first processed separately and then combined with all previous levels and checked for combinations provided there is at least one node from both the sets of previous levels and the new level. The function *divIntoKLevelSets(T)* divides T into sets of consecutive k levels, *resetLevlsSMs(Levels, streaming multiprocessors)* resets the streaming multiprocessors marking all of them as available and marks all the k-level sets as new, and ready to be processed. The function *testCon(Comb)* checks if the subgraph induced by the nodes in *Comb* is connected or not.

4.3.6 Storing Graphs on GPUs

Adjacency data of the input graph required for computation is stored on the GPU. Memory hierarchy of the GPU consists of global memory, shared memory, constant memory, texture memory and registers. The size of each of the above mentioned types of memory varies according to the system, and a comparison is given in Table 4.1. The global memory is the largest and also has the highest access latency. The on-chip shared memory, which is further divided into 16 (or 32) banks, has significantly faster

access compared to the global memory. But, when data is accessed from the same bank, significant performance loss occurs due to bank conflicts (the only exception being the case where all the threads access the same element leading to a broadcast.)

Model #	Cores	Global Mem. (GB)	Sh. Mem. (KB)	# of Mem. Banks	Comp. Cap.
C1060	240	4	16	16	1.3
C2050	448	3	48	32	2.0
C2070	448	6	48	32	2.0

Table 4.1: Architecture Comparison of Different Nvidia GPUs

For a graph $G = (V, E)$ with $|V| = n$, the size of adjacency matrix is n^2 bits, where each edge is stored using a single bit. Now, to fit the graph in memory, the size required must be less than or equal to the space available. Therefore, for storing graphs using adjacency matrix data, the following equation must be satisfied

$$n^2 \leq S_{mem} \tag{4.1}$$

where, S_{mem} is the size of the memory in bits.

For undirected graphs, values (i, j) and (j, i) are same. So, storing only the Upper Triangular Matrix (UTM) of the adjacency matrix is enough. Therefore, in this case,

$$\frac{n \times (n+1)}{2} \leq S_{mem} \tag{4.2}$$

As all the values of $(i, j) = 0$ when $i = j$, using the Strictly UTM representation (S-UTM) (i.e. without the data on the diagonal), size of the largest graph increases by 1. Using Equation (4.1) and Equation (4.2), the largest graph that can be kept in the shared memory and global memory for the different systems is given in Table 4.2.

Model #	Shared Mem. Adj Mat	Shared Mem. S-UTM	Global Mem. Adj. Mat	Global Mem. S-UTM
C1060	362	512	$185,363$	$262,144$
C2050	627	887	$160,529$	$227,023$
C2070	627	887	$227,023$	$321,060$

Table 4.2: Maximum size of graphs on different GPUs

4.3.7 Handling Larger Graphs

For graphs that fit in the shared memory, algorithms described in (Chatterjee et al., 2012) can be used to do the computations. In this Section we consider graphs of larger sizes. The following assumptions can be made for the properties of such graphs:

- For a given graph $G = (V, E)$, the adjacency information for G does not fit in the shared memory i.e.,

$$S_G \geq S_{SM} \tag{4.3}$$

  where, S_G is the size of G in bits using the most efficient data structure, and S_{SM} is the size of the shared memory in bits.

- G can be preprocessed on the CPU and split into chunks taking into consideration consecutive levels of BFS-tree T of G i.e.,

$$G \Rightarrow G_1 \cup G_2 \cup \cdots \cup G_i \tag{4.4}$$

- Let $G = (V, E)$, where $V = \{v_1, v_2, \cdots, v_n\}$. Starting with node v_i, G can be split into subgraphs, where $G = G_{i1} \cup G_{i2} \cup \cdots \cup G_{ij}$. Let, $S = \{s_1, s_2, \cdots, s_n\}$, where

$$s_i = \sum_{m=1}^{\alpha_i} C_{im} \tag{4.5}$$

where, $C_{im} = 1$ if $S_{G_{im}} > S_{SM}$, else $C_{im} = 0$, and α_i is the number of splits possible starting with node v_i. Therefore, to ensure the minimum number of chunks with size greater than the shared memory is chosen s_i is selected, where $s_i = \text{Min}\{\text{S}\}$, and the output of the preprocessing is G_{i1}, G_{i2}, $\cdots$, G_{im}.

- To ensure that the fragmentation is the least i.e., minimum wastage of shared memory (for the chunks that actually fit in the shared memory), the following condition must be satisfied

  $$Min\{S_{SM} \times P - \sum_{m=1}^{\alpha_i} S_{G_{im}}, \forall S_{G_{im}} \leq S_{SM}\}$$

  where, P is the number of streaming multiprocessors or modules in the Nvidia system concerned.

Algorithm 2: Splitting G on the CPU

```
Input: Graph G
Output: Graphs G1,G2,···, Gi,
where G = G1 ∪ G2 ∪ ··· ∪ Gi
begin
    {CCi} ← findConnectedComponents{G};
    foreach CCi do
        while CCi_size ≥ SSM & vi ∈ CCi(V), where ∃ vi ∉ processed do
            {Ti} ← createBFStree {CCi, vi};
            {Li} ← divIntoConsLevelSets(Ti);
            if Li_size ≤ SSM ∀Li then
                Output ← L1, L2, ···, Li;
                Mark CCi processed;
                break;
            end
            Mark vi processed;
        end
        if CCi ∉ processed then
            Output ← CCi;
        end
    end
end
```

Algorithm 2 splits input graph G into sets of consecutive level nodes using Breadth-first search property and considering each connected component of G separately. After

splitting the graph using Algorithm 2, certain sets of nodes or chunks might not fit in the shared memory. The adjacency information for the nodes in such sets is kept in the global memory. Therefore, now the threads in the GPU access data from both shared and global memory.

Due to the difference in access latency of global and shared memory, threads operating on data stored in the global memory might require significantly more time to do the computations. Therefore, if the computations on the data stored in the shared memory and the global memory are performed sequentially, it would take more time than an intelligent scheduling of the computations on the streaming multiprocessors so that, at any instant of time during execution, active warps operate on adjacency data stored in both shared and global memory. However, it might be the case for specific instances of graphs, where none of the sets of nodes or chunks fit in the shared memory, and all the sets are in the global memory. In that case, operations on the data can be performed accessing the global memory efficiently taking advantage of memory coalescing and avoiding partition camping.

The above idea can be illustrated using the following example. Let the total number of sets of data to be computed on be ψ. Also, let the number of sets of data that fit in the shared memory be given by ψ_s and that in the global memory be ψ_g, where $\psi_s + \psi_g = \psi$. Suppose it takes on an average τ_s units of time to operate on a set of data stored in shared memory and τ_g be the corresponding value for data in global memory. If $\psi_s \leq 30$ and the operations on the data stored in shared memory and that in the global memory are performed one after the other as mentioned above, the total time taken for execution, is given by $\tau_s + \psi_g \times \tau_g$, since the data in the shared memory is accessed in parallel and that in the global memory is accessed in a sequential manner. Therefore, for the general case, the total time taken to compute

on all the sets of data is given by the following equation:

$$\tau_t = \mu \times \tau_s + \psi_g \times \tau_g \tag{4.6}$$

where, τ_t is the total time and $\mu = \lceil \frac{\psi_s}{30} \rceil$. An efficient scheduling of the active warps would minimize the value given by Equation (4.6).

4.3.8 Scheduling threads to operate on data chunks

After the given graph G is split into chunks by using Algorithm 2, the data corresponding to these are stored either in the shared or global memory, as required. Now, blocks of threads executing on the streaming multi-processors are scheduled to operate on the data. The objective is to minimize the total time of execution as discussed in the previous Section. This can be done by scheduling the operation on the data for each of the chunks on the available GPU streaming multiprocessors, so that the time required is minimum. This problem is equivalent to the Makespan Scheduling problem, and is NP-hard (Grabowski & Wodecki, 2004).

The Makespan Scheduling problem is defined as follows: Given Θ machines for scheduling, indexed by the set $M = \{1, \cdots, \Theta\}$, and η jobs, indexed by the set $J = \{1, \cdots, \eta\}$, where job j takes $p_{i,j}$ units of time if scheduled on machine i, and J_i is the set of jobs scheduled on machine i. Then $l_i = \sum_{j \in J_i} p_{i,j}$ is the load of machine i. The maximum load $l_{max} = \max_{i \in M} l_i$ is called the makespan of the schedule.

In the context of the computations on the GPU, the machines are the modules i.e., streaming multiprocessors, and all modules are identical, and the processing time of the jobs are the size of the chunks. Therefore, the value of $p_{i,j}$ and $p_{k,j}$ are same. But, the problem is still NP-hard even if there are only two identical machines, which is much simpler than the problem at hand with 30 identical streaming multiprocessors.

An example of the Makespan Scheduling for the given problem is illustrated us-

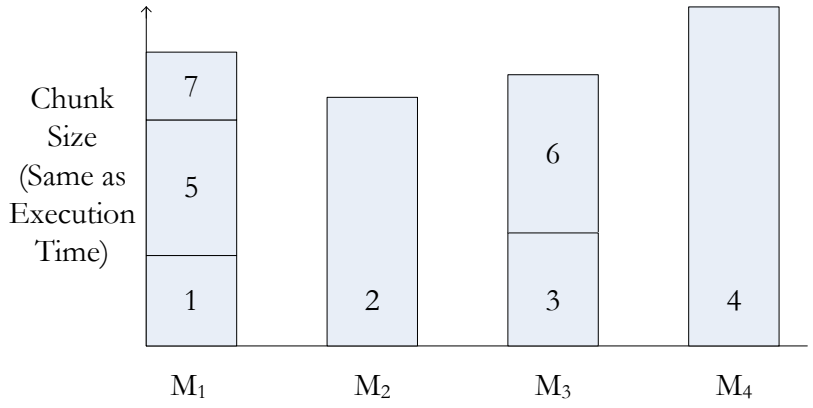


Figure 4.2: Executing chunks on GPU cores: Makespan scheduling

ing the diagram in Fig. 4.2. Here, M_i's represent the machines i.e., the streaming multiprocessors (just 4 are shown for simplicity), and the rectangles numbered from 1 through 7 represent the jobs i.e., computations to be performed on the chunks. Threads in each streaming multiprocessor operates on the chunks and the time required is proportional to the size of the chunks. In this case, while chunks 1, 5 and 7 are computed sequentially by SM M_1, SM M_2 operates on chunk 2, SM M_3 operates on chunks 3 and 6 and SM M_4 operates on chunk 4, all in parallel. However, as mentioned earlier, assigning chunks to the streaming multiprocessors so as to minimize the maximum makespan is NP-hard.

4.3.9 Results

Experiments are performed using both CPU and GPU. The CPU consists of quad-core 2.27 GHz Intel Xeon processors, with 12 GB memory. The GPU used for the experiments is Nvidia C1060 card, with 4 GB device memory and 16 KB shared memory per multiprocessor . The problem of finding the number of connected subgraphs of size k from a graph of size n is solved on the GPU for different values of n and k by storing the adjacency information of the graph in both the shared memory and the global memory, while using both a single streaming multiprocessor and also all

available streaming multiprocessors. The number of threads in each of the streaming multiprocessors considered is limited to 32, which is equal to the *Warp-size* (NVIDIA Corporation, 2010). The data set under consideration is relatively small, comprising of graphs consisting of approximately 300 nodes. Therefore, we do not need too many threads to work on it. If we use more than 32 threads per block, then the number of blocks would decrease and many of the streaming multiprocessors out of the available 30 would go unused. Also, since there are 8 processor cores per streaming multiprocessor, and threads are executed in warps of size 32, keeping the number of threads at 32 per thread block also helps minimize the context switching overhead that may exist.

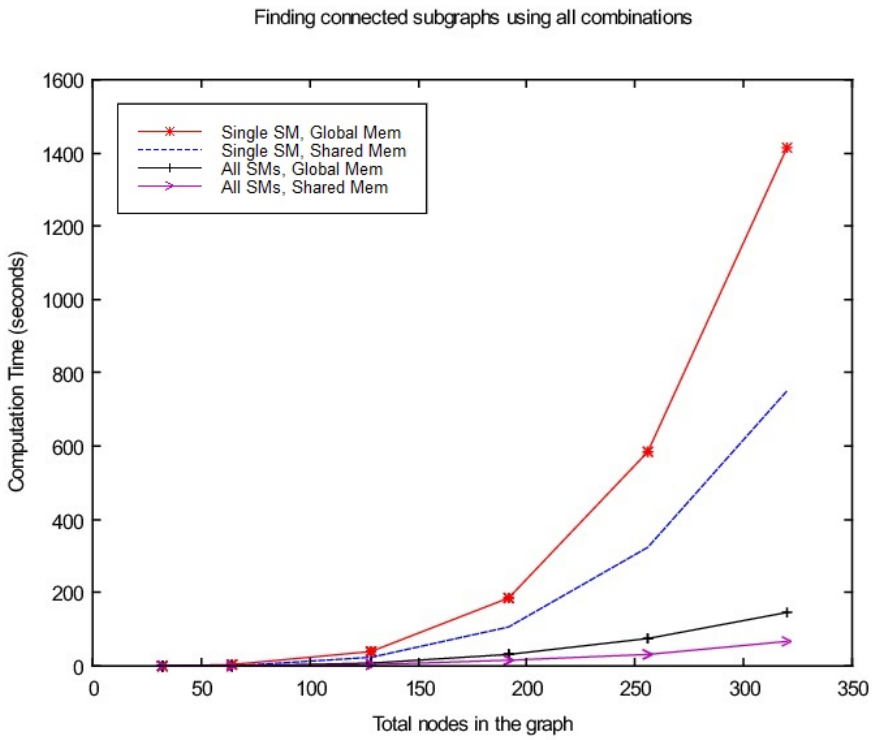


Figure 4.3: Evaluating all combinations for $k = 3$ with 32 threads in each streaming multiprocessor (SM), for data stored on both shared and global memory

Fig. 4.3 plots the timings for evaluating all the combinations for the graph kept on both the shared and global memory while using both single and all available streaming

multiprocessors. The plots are as expected, with the timings for the shared memory better than those compared to the global memory. Also, by using all the streaming multiprocessors as compared to using just a single streaming multiprocessor, more threads are available which leads to a better performance.

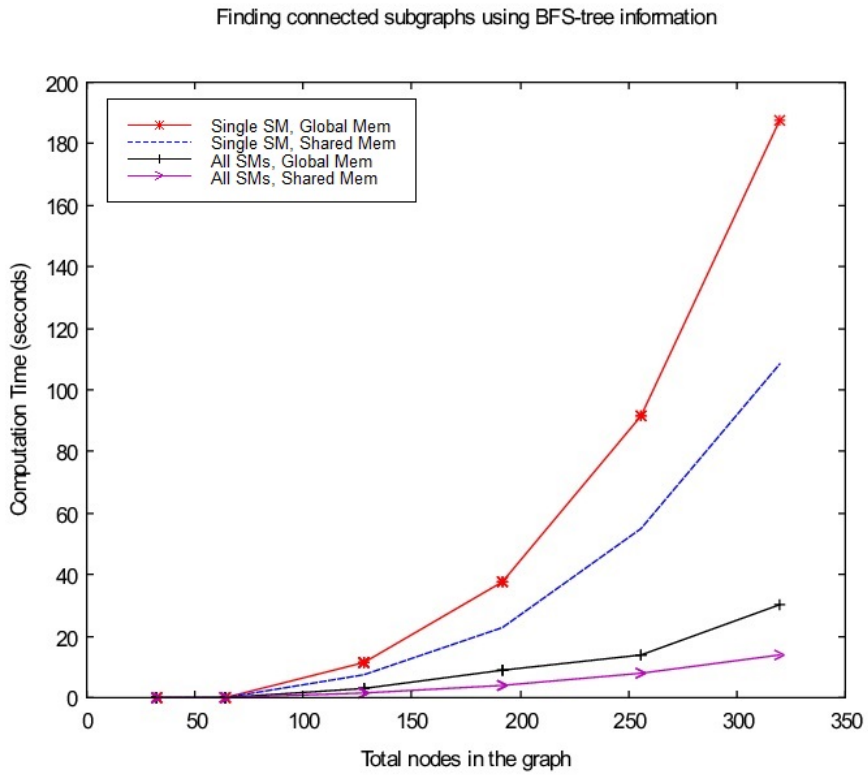


Figure 4.4: Evaluating reduced number of combinations using BFS-tree information for $k = 3$ with 32 threads in each streaming multiprocessor (SM), for data stored on both shared and global memory

Fig. 4.4 plots the timings for the graph kept on both the shared and global memory considering the BFS-tree topology information while using both a single streaming multiprocessor and all the available streaming multiprocessors. The number of computations and resulting computation times are greatly reduced in this case as compared to the previous case (Fig. 4.3) where all the combinations of the nodes are tested.

Fig. 4.5 plots the timings for the graph kept on the shared memory, using all

available streaming multiprocessors, and evaluating both all and reduced number of combinations thereby comparing the previous two cases. It is clear from the Figs. 4.3 − 4.5 that the calculations done using the BFS-tree topology information while keeping the adjacency information on the shared memory and utilizing all available streaming multiprocessors using a large number of threads is the most efficient approach.

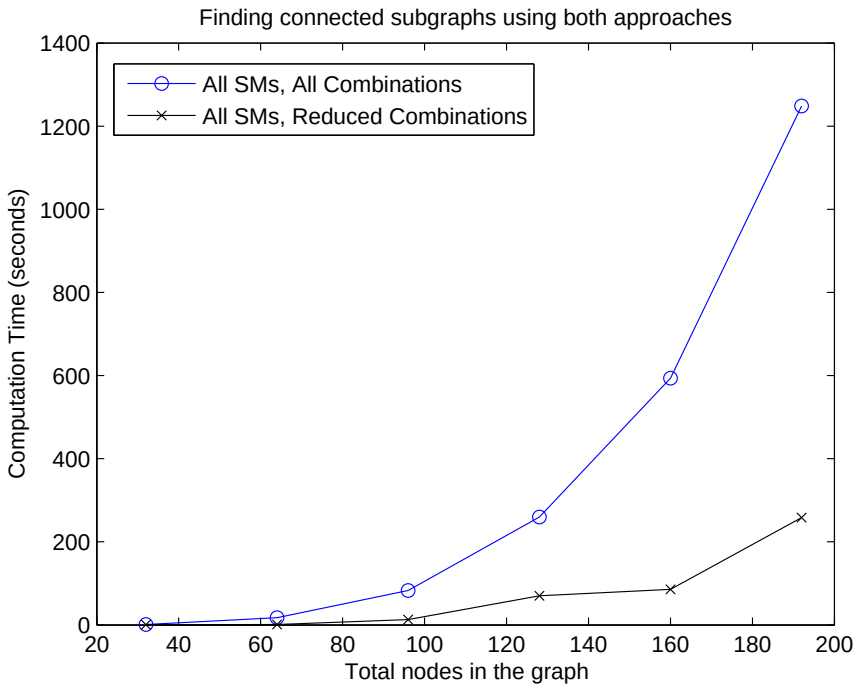


Figure 4.5: Evaluating all combinations and reduced combinations for $k = 4$ with 32 threads in each of the streaming multiprocessors (SMs), for data stored on shared memory

The connected subgraph counting problem is also solved on the CPU for subgraphs of size 3. The CPU implementation uses the same data structures and computation techniques as those employed on the GPU. However, the CPU uses a single thread for execution of the program compared to a large number of threads on the GPU. One of the major techniques suggested in the counting problems is to consider the breadth-first search tree information to exclude unnecessary computations. The graph data

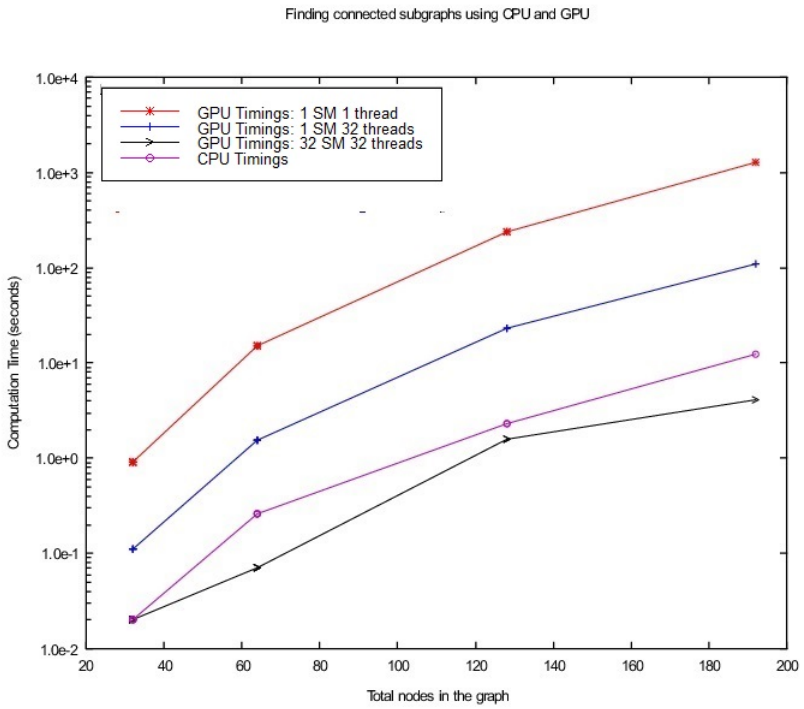


Figure 4.6: Comparing timings between CPU and GPU for $k = 3$ using BFS-tree information

that is finally used for computation is rearranged according to the BFS-tree levels. Therefore, due to the reordering of the data, the potential for cache line optimization also increases. Hence, the proposed techniques are not only useful for the GPU but also help in optimizing execution on the CPU too. Fig. 4.6 provides a comparison of the timings for the implementation on the CPU and GPU. The plots, using a base 10 logarithmic scale for the y-axis, in Fig. 4.6 show the speedup gained on the GPU as compared to the CPU while varying the number of threads on the GPU.

Experiments were also performed using the data available on the Stanford Network Analysis Project (Leskovec et al., 2009). Using reasonably larger graphs of size ranging from $5,000$ to $25,000$ nodes, it can be observed, as shown in Fig. 4.7, that the computation on the GPU attains a 10 times speedup as compared to that on the CPU. In this case, the graph data is stored on the global memory and the graphs are sparse thereby having less number of combinations to be checked. This proves that

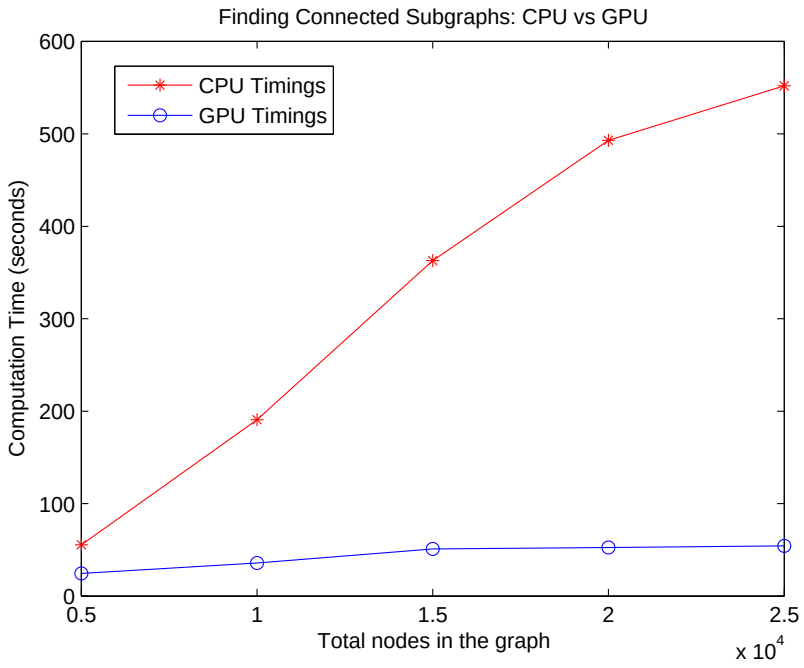


Figure 4.7: Comparing timings for larger graphs

the proposed solution is applicable for large graph instances too, which is essential for analysis of social networks and other real world datasets.

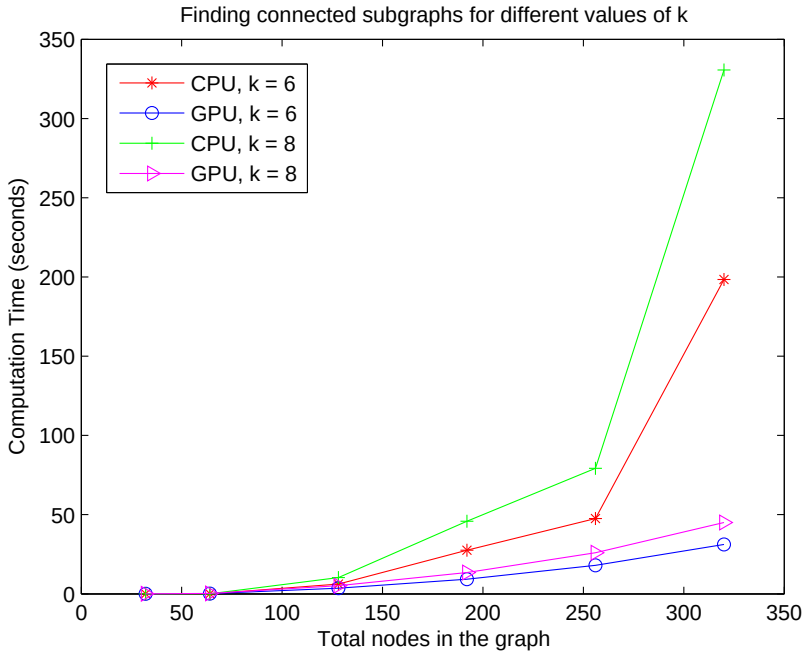


Figure 4.8: CPU and GPU timings for larger values of k

In the previous experiments, only subgraphs of size 3 and 4 are considered. For larger values of k, the total number of combinations to be checked for specific properties in the graphs induced by the nodes in consideration would increase, provided $k \leq n_k/2$, where n_k is the number of nodes in any set of consecutive k-levels. Since more computation would be required for larger values of k, the speedup is better on the GPU as compared to that on the CPU, as shown in Fig. 4.8.

4.4 Counting cliques and independent sets

Using similar approaches as adapted in the connectivity testing algorithm, graph problems like finding total number of cliques of size k and total number of independent sets of size k can also be solved. For finding the total number of cliques of size k, only nodes in adjacent levels T needs to be considered, as given in Algorithm 3.

Algorithm 3: Counting number of k–cliques

```
Input: BFS-tree T of graph G
Output: Total count of cliques of size k
begin
    {L_i} ← divIntoLevels(T);
    {L_i} ← markLevelsNew({L_i});
    TotalCount ← 0;
    TotalLevels ← 0;
    while L_{i.Status} ∈ {L_i} = New do
        curLvl ← L_i;
        TotalCount ← TotalCount + testClique(GenNxtComb (curLvl));
        TotalLevels + +;
        if TotalLevels > 1 then
            curLvl ← L_i ∪ L_{i−1};
            TotalCount ← TotalCount + testClique(GenNxtComb
              (curLvl));
        end
    end
end
```

For finding total number of independent sets in G, it's complement say G' is taken,

and then a BFS is performed on G' to get T', as given in Algorithm 3. Finding cliques of size k in T' is equivalent to finding independent sets of size k in G, as given in Algorithm 4.

Algorithm 4: Counting independent sets of size k

Input: Graph $G(V, E)$
Output: Total count of independent sets of size k
begin
    $\{G'\} \leftarrow$ `FindComplement(`G`);`
    $\{T'\} \leftarrow$ `BFS-TreeGenerate(`G'`);`
    $TotalCount \leftarrow$ `Algorithm3(`T'`);`
end

4.5 Generating combinations for testing in graphs

The problem we are trying to solve is to verify certain properties in subgraphs of a specific size k for a given graph $G = (V, E)$, where $|V| = n$. Examples of such properties include connectivity, formation of cliques and formation of independent sets. To solve the above mentioned problem, sets of k nodes from n available nodes are to be chosen, and then verified for the required property. This leads to generation of combinations of all possible sets of nodes to check for the correct result. Following are some of the approaches that can be adopted to achieve the desired outcome.

4.5.1 Sequential approach with pre-computed combinations

A basic method to test for all combinations is to generate the same in the pre-processing stage and store them. During the actual computation, the combinations can be retrieved and tested for the desired property. The major drawback with this approach is the amount of memory required just to store the pre-computed combinations. For a graph with n nodes and for subgraphs of size k, there would be ${}^nC_k$ tests in total. Now, each of the combinations consist of k values, each of which

required $log(n)$ bits for storage. Therefore, the total storage required for such an instance is ${}^nC_k \times k \times log(n)$ bits.

4.5.2 Sequential approach with combinations generated on the fly

The previous approach can be improved by not storing the combinations. During the testing phase, the combinations can be generated one-by-one sequentially according to the lexicographical order (Mifsud, 1963). This method requires storing the previous combination to generate the next one. Therefore, the space required for storage is $2 \times k \times log(n)$ bits. But this is a sequential approach and dividing work among multiple available threads cannot be done efficiently using this technique.

4.5.3 Naïve division of combination testing among available threads

A naïve approach that would help divide the work among threads is to generate combinations based on the starting numbers, and split the work according to thread id's matching node numbers. There would be $n - k + 1$ such starting combinations, and hence the same number of threads are required. In such a case the number of threads is in the order of n. If there are more number of threads available, then combinations can be generated with slight modifications, say considering first 2 nodes different for each thread thereby utilizing n^2 threads. However, this approach leads to uneven distribution of work among threads, with threads having id numbers in the beginning doing more work than the ones with lager id numbers.

4.5.4 Equal work division among all available threads

The easiest way to ensure all threads do same amount of work is by calculating the total number of tests possible and then dividing the work among all available threads. By taking this approach all threads are now responsible for an equal amount of work (some threads might have to do a single test more if the number of threads does not

divide the total work evenly). Now, the naive approach to solving the problem using the above technique is to generate all the possible combinations to test and store them in a data structure, and let the threads access the specific combinations for testing. However, this approach would require a lot of space to store all the combinations in the first place as discussed before.

Therefore, the ideal approach would be able to generate the combinations during runtime and perform the testing. Since, threads would require random lexicographical combinations to work on, generating the same efficiently is important. Generating a specific combination using the index in the lexicographic order can be done efficiently (Buckles & Lybanon, 1977). There exists a mapping from natural numbers i.e., indices in the lexicographic order to combinations, and this methodology is also known as *combinadics* (McCaffrey, J., 2004).

Using combinadics, lexicographic ordering for any index can be found from a single set of elements (which can be node numbers in the case of graph problems.) However, while using BFS-tree information for testing reduced number of combinations, there are additional constraints with respect to the number of sets of elements and number of items chosen from each. Generating combinations for the sets of levels in the BFS-tree involves restrictions like including at least one node from the first level in the chosen set to avoid redundant checking, except in the case of the last set of levels. A naïve method to use combinadics would be to calculate the combination considering all nodes from the given set of levels without restrictions, and then make sure at least one node belongs to the first level, else continue generating the next valid combination.

4.6 Counting Triangles in graphs

Finding and counting triangles in graphs is a fundamental problem, and can be solved using GPUs. It is a common graph-mining task, as the number of triangles is closely connected with estimating the clustering coefficients and the transitivity ratio of the graph. Counting the number of triangles has other applications too, such as spam detection (Becchetti et al., 2008). In the context of social networks, triangles have a natural interpretation: friends of friends tend to be friends, and this can be used in potential friend suggestion, as shown in Fig. 4.9. Counting triangles in large graphs is described in (Becchetti et al., 2008).

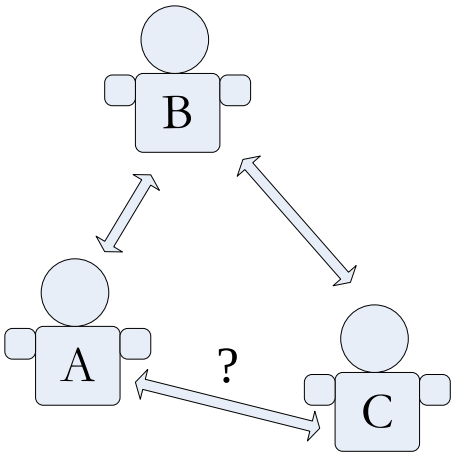


Figure 4.9: Triangles in Online Social Networks

A triangle $\Delta = (V_\Delta, E_\Delta)$ of a graph $G = (V, E)$ is a three node subgraph with $V_\Delta = \{u, v, w\} \subseteq V$ and $E_\Delta = \{u, v\}, \{v, w\}, \{w, u\} \subseteq E$. Let $\vartheta(G)$ denote the number of triangles in graph G. It can be noted that an n-clique has exactly ${}^nC_3$ triangles i.e., $\vartheta(n\text{-clique}) = {}^nC_3$.

Operations on graphs for finding triangles can be of two types: a) *counting*: finding the number of triangles, and b) *listing*: identifying the nodes forming the triangle, and reporting the same. In this section we discuss the algorithm and implementation

methodologies for *counting* triangles in graphs. Optimization techniques that help in increasing the efficiency are discussed in the later Sections. Algorithm 5 finds triangles in a given graph G. It can be noted that the function *GenNxtComb*() finds combinations of 3 nodes for further testing. When *GenNxtComb*() is called with the parameter *bothLvls*, it returns combinations containing 3 nodes from the set of consecutive levels, out of which at least 1 is from the $firstLvl$. This restriction eliminates duplicate checking for any combination of nodes. Fig. 4.10 shows the grouping of adjacent levels of a sample BFS-tree required for counting the number of triangles.

Algorithm 5: Counting number of triangles in G

```
Input: BFS-tree T of graph G
Output: Total number of triangles in G
begin
    {L_i} ← divIntoConsecutiveLvlSets(T);
    TotalCount ← 0;
    foreach L_i do
        TotalCount ← TotalCount + testTriangle(GenNxtComb(firstLvl));
        TotalCount ← TotalCount + testTriangle(GenNxtComb(bothLvls));
        if L_i is the last set then
            TotalCount ← TotalCount +
            testTriangle(GenNxtComb((SecondLvl)));
        end
    end
    Output ← TotalCount;
end
```

Apart from counting the number of triangles that exist in a given graph, Algorithm 5 can also check if a graph is *triangle-free*. A triangle-free graph is an undirected graph in which no three vertices form a triangle of edges. Triangle-free graphs are equivalent to graphs with *clique number* ≤ 2, or graphs with *girth* ≥ 4.

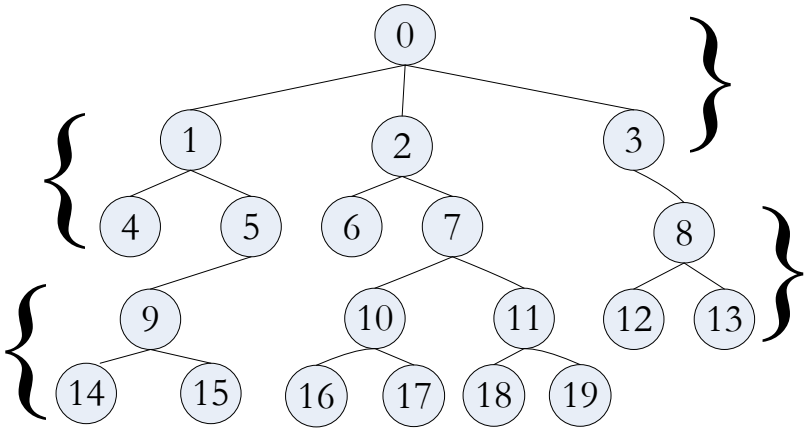


Figure 4.10: BFS-tree level grouping for finding triangles

4.6.1 Avoiding Partition Camping While Accessing Data for Graph Problems

Partition camping takes place due to memory access patterns among different active warps. For solving the counting problem on triangles as given in Algorithm 5, different sets of levels are computed upon by different warps being executed on available streaming multiprocessors. Due to the pattern of the data being accessed, adjacent sets of levels have some shared levels (1 in the case of counting triangles, k-cliques, k-independent sets and k - 1 in the case of connected subgraphs of size k.) Therefore, during computation, due to the amount of shared data, threads from different active warps might need to access the same information leading to accessing the same partition in the global memory, thereby causing partition camping.

The analysis of the issue can be illustrated using the example of the BFS-tree given in Fig. 4.10. Now, for finding triangles, as given in the Algorithm 5, different warps would work on different adjacent level sets (ALS) at the same time. Let us assume, warp W_i operates on a set of levels given by ALS_i and is executed on the streaming multiprocessor SM_i. For the graph in Fig. 4.10, let us consider the second and the

third sets of levels. When threads operating on these sets from different warps access the data related to the common nodes (numbered 4 to 8) simultaneously, partition camping occurs. The potential for such partition camping results from using a single adjacency matrix for the entire graph as the data structure, as shown in Fig. 4.11. It must be noted that the same problem would arise by using other data structures too, provided the entire data is stored together.

Keeping relevant data for the adjacent level sets separately in different partitions solves the above issue. For example, the data in the regions of intersection i.e., between data for ALS_1 and ALS_2, and also between ALS_2 and ALS_3 would have to be duplicated. Also, the design would be efficient if the data for a specific set of levels do not span across partitions. Therefore, the width of the sets of adjacency information data must be a maximum of 256-bytes. An arrangement of the data using the above mentioned idea is shown in Fig. 4.12.

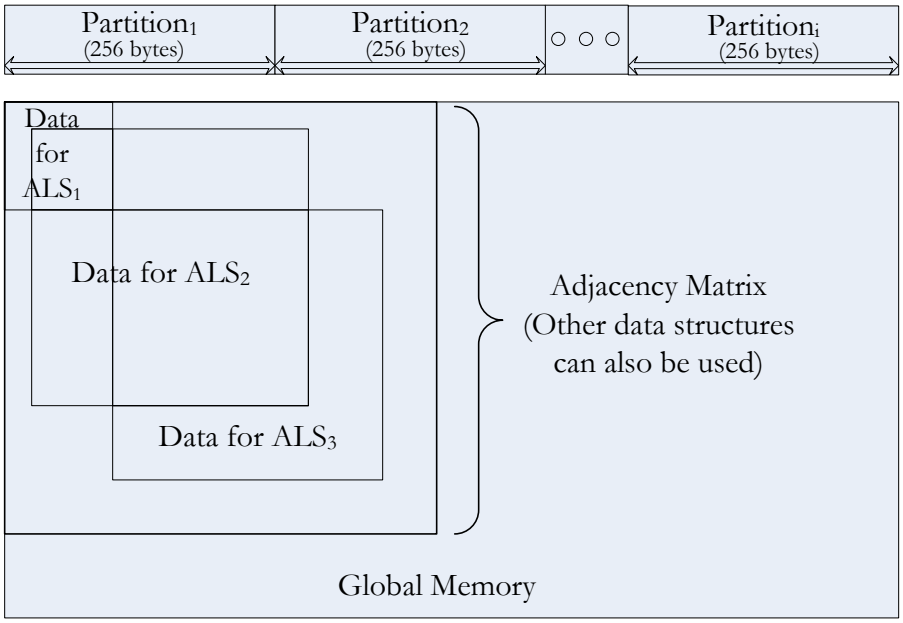


Figure 4.11: Data structure with potential for partition camping

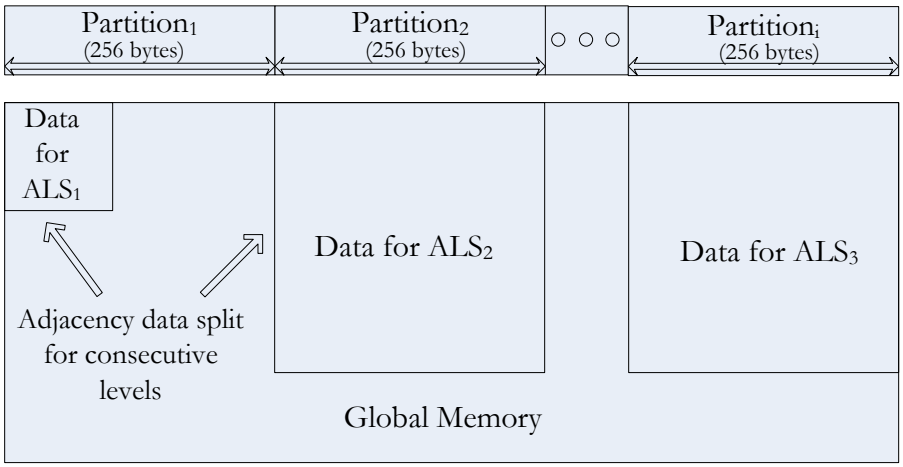


Figure 4.12: Storing redundant information for avoiding partition camping

4.7 Experimental results

The triangle counting problem as described in Algorithm 5 is implemented using both CPU and GPU. The CPU consists of quad-core 2.27 GHz Intel Xeon processors, with 12 GB of memory. The GPU used for the experiments is Nvidia C1060 card, with 4GB of memory. The CPU implementation is performed using a single thread.

Triangles are counted in graphs of sizes ranging from 200 to 2000 nodes. The timings for the CPU and GPU implementation are plotted and compared in Fig. 4.13. For implementing triangle counting using Algorithm 5, a BFS-tree for the input graph is required which is generated using Algorithm 2. Therefore, the timings for counting triangles include the executing time for both Algorithms 2 and 5.

For smaller size graphs, due to overhead in transferring data from the host i.e., the CPU memory to the device i.e., the GPU memory, the timings are almost similar. But, as can be observed from the plots in Fig. 4.13, the performance of the GPU increases with the number of nodes in the graph. For 1000 nodes or more, there is 5–6 times improvement in the timings of the GPU as compared to the CPU.

The triangle counting algorithm is then implemented on the GPU using modified data structures to incorporate the usage of available primitives like memory access

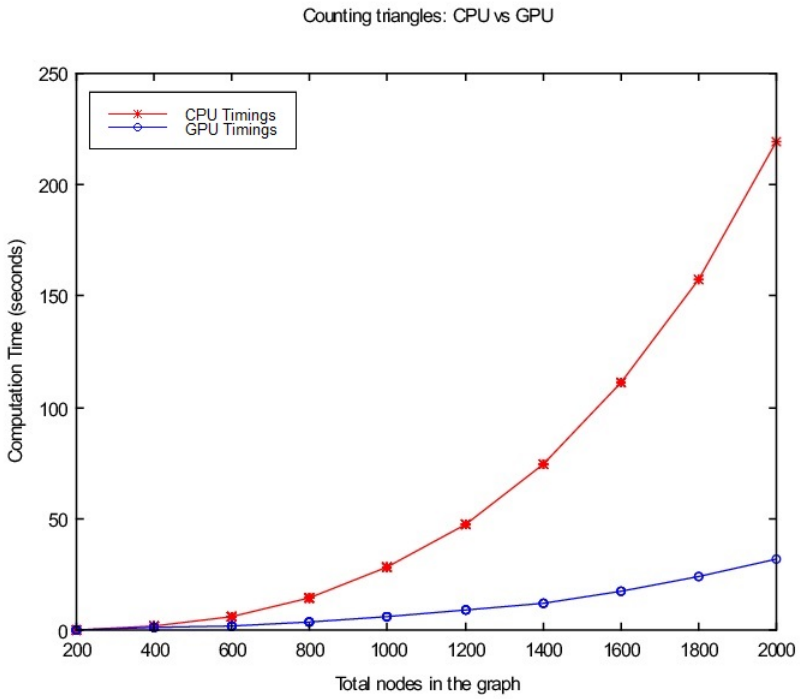


Figure 4.13: Comparing timings for counting triangles using CPU and GPU

coalescing and avoiding partition camping, as discussed in the earlier Sections. The timings for the implementation using naïve data structures and modified ones with redundant data are plotted for comparison in Fig. 4.14. The experiments are done on input graphs of size ranging from 200 to 2000 nodes.

As can be observed from the plots in Fig. 4.14, the performance of the GPU increases when making use of the available primitives and approximately 6–8 % performance gain is achieved over the naïve implementation. Therefore, the triangle counting problem achieves high speed-up from being solved on the GPU as compared to the CPU, and additionally the efficiency of the naïve implementation is further improved by using the available memory access primitives for the global memory effectively.

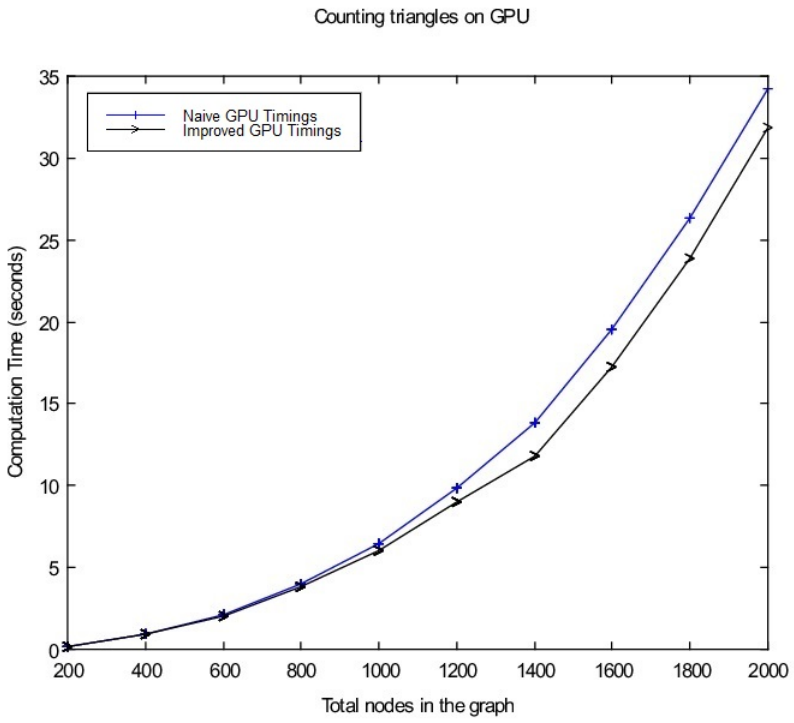


Figure 4.14: Counting triangles using global memory with memory access coalescing and avoiding partition camping

4.8 Summary

In this chapter, algorithms to solve several graph problems on a parallel GPU architecture are proposed. The major focus is utilizing faster shared memory of the GPUs and devising data structures to represent graphs in these small memory modules. Methods to generate combinations to efficiently divide the work among threads belonging to both single and multiple streaming multiprocessors are developed. In addition, techniques to reduce computations by using breadth-first search tree and exploiting topology information are discussed.

For studying the properties of graphs, generating combinations of nodes is required. The overhead of combination generation can be significant, both from the required time and space perspective. Therefore, techniques to efficiently generate combinations of nodes to be tested on graphs have been discussed in this paper.

An algorithm for triangle counting in a given graph utilizing all the above mentioned primitives is analyzed and implemented, and the results are reported. As evident from the experimental results, by properly utilizing the above methodologies, access time for retrieving data from the global memory is effectively reduced and improves the performance by a significant amount as compared to the naïve implementation. The triangle counting algorithm is also implemented on the CPU, and comparative analysis of the results are done with those of the GPU. Certain sections of this chapter are adapted from our research papers (Chatterjee et al., 2013a) (Chatterjee et al., 2013b).

Chapter 5

Analysis on large data sets

Studying the properties of Online Social Networks (OSNs) and other real world graphs have gained importance due to the large amount of information available from them. These large graphs contain data that can be analyzed and effectively used in advertising, security and improving the overall experience of the users of these networks. However, the analysis of these graphs for studying specific properties requires combinatorially explosive number of computations. Compute Unified Device Architecture (CUDA) is a programming model available from Nvidia for solving general-purpose problems using the massively parallel and highly multi-threaded Graphics Processing Units (GPUs). Therefore, using GPUs to solve these types of problems is appropriate. In addition, due to the properties of real-world data, the graphs being considered are sparse and have irregular data dependencies. Hence, using efficient techniques to store the graph data for initial preprocessing and final computation by taking advantage of heterogeneous CPU-GPU systems can address these issues.

In this chapter, we are interested in studying different properties of these real-world entities that transform into the following graph problems: a) identifying a missing edge, which when added would result in maximum increase in the number of triangles, b) identifying an existing edge whose removal would result in the maximum decrease in the number of triangles, c) identifying an existing edge whose removal would increase the number of connected components in the graph. In this chapter, we develop and implement algorithms to solve the above problems using both CPU and GPU. Specifically, given a graph $G = (V, E)$, we provide algorithms for the

following: a) find $(v_i, v_j) \notin$ E, such that $\Delta_f - \Delta_c$ is maximized, where Δ_f and Δ_c are the number of triangles in $G_m = (V, E \cup (v_i, v_j))$ and G, respectively, b) find a $(v_i, v_j) \in$ E, such that $\Delta_c - \Delta_f$ is maximized, where Δ_f and Δ_c are the number of triangles in $G_m = (V, E \setminus (v_i, v_j))$ and $G = (V, E)$, respectively, c) find a $(v_i, v_j) \in$ E, such that $\phi_m > \phi_c$, where ϕ_m and ϕ_c are the number of connected components in $G_m = (V, E \setminus (v_i, v_j))$ and $G = (V, E)$, respectively. We implement the algorithms using a GPU and achieve a 10 $\times$ speedup as compared to a sequential implementation. Thereafter, we design a heuristic for finding an edge whose existence would result in the maximum increase in the number of triangles. The heuristic is implemented and the results are reported and compared to those of the regular algorithm on the GPU.

5.1 Introduction

The recent growth in Online Social Networks (OSNs) and the existence of other large graphs have provided a plethora of exciting information (Twitter Statistics, 2014) (Facebook Statistics, 2014) (LinkedIn Press Center, 2014). Many new questions arise from psychological and anthropological point of view, where valuable insight can be gained from analyzing the properties of these graphs. Advertising and security can also be improved from the study of the same (Yu et al., 2006). Study of the following characteristics in the graphs representing OSNs can be useful: a) identifying missing connections whose future existence would result in maximum increase in 3-way mutual relationships, b) identifying existing connections whose elimination would result in maximum decrease in 3-way mutual relationships, c) identifying existing connections whose elimination would increase the number of different groups of people. The above problems require testing for different combinations of entities and are computationally expensive. For example, for an OSN with n users, testing for triangles among the users using a naïve algorithm requires n^3 comparisons; for large graphs, this would result

in a huge amount of computation. Therefore, in this chapter we use techniques based on breadth-first search to reduce the solution space and perform faster calculations. Dividing the data based on the breadth-first search tree properties also addresses the issue of irregular data dependencies by grouping related data together. Also, the easy availability of GPUs and programming models like CUDA to solve general purpose problems using the same has steered research in this direction (NVIDIA Corporation, 2010). In this chapter, we design algorithms to solve the above problems and implement them using heterogeneous CPU-GPU systems. Since the problems discussed above requires huge amount of computation, we also propose a heuristic to provide an approximate solution to the same. Instead of performing computation on the entire data, consisting of millions of nodes for real-world graphs, the heuristic takes into account some basic properties of the graph and reduces the problem space by a significant amount thereby providing a much faster solution.

5.2 Related work

Previous work has been done on analyzing OSNs and also computing on large data using GPUs. There has been research done on identifying clusters in social and information networks. Results show well-connected clusters or communities are small in size and have few connections to other entities in the network (Leskovec et al., 2009). In this chapter we have independently shown similar properties in graphs by processing the breadth-first search tree of the same. The different levels of the BFS tree follow patterns that are similar to those of the clusters or communities mentioned above, and are relevant for performing the analysis of the graphs using GPUs.

Advertising and marketing can benefit from recommendations provided by connections in social networks (Leskovec et al., 2007). The problems discussed in this chapter help identify possible connections between entities that would result in larger

clusters and a larger target domain and are therefore relevant for analysis of OSNs and other large graphs.

Counting triangles in massive graphs with the focus on local triangle counting has been addressed in previous research (Becchetti et al., 2008); the algorithms provide an estimate of the number of triangles associated with each node in a given graph and spamming activity in graphs can be detected based on the extracted information. In this chapter, we discuss techniques to keep count of the number of triangles taking into account the dynamic nature of graphs where edges can be added or deleted over time, without performing the entire counting calculations again.

Our previous work deals with counting triangles in graphs (Chatterjee et al., 2013a). The research focus is on identifying techniques to efficiently store the data to be computed on and provide algorithms to perform the computations effectively (Chatterjee et al., 2013b). In this chapter, we introduce algorithms that are suited to the properties of large real-world data sets; our analysis shows techniques to effectively leverage the GPUs for computation on such data.

Work has also been done on counting triangles in massive graphs. Techniques to speed-up the counting has been studied in detail (Tsourakakis et al., 2009). Counting triangles using the MapReduce framework has also been analyzed (Suri & Vassilvitskii, 2011).

Properties of OSNs have been studied, and interesting observations have been made on real-world data (Mislove et al., 2007) (Viswanath et al., 2009) (Mislove et al., 2010). However, instead of simply identifying existing properties, in this chapter we focus on studying the behavior of the network when the graph changes, which resembles more closely the real-world scenario and behavior of large graphs.

5.3 Triangle completion problem

The growth of OSNs is usually measured by the number of users i.e., the number of nodes from the graph perspective. OSNs follow the power-law graph, with a high probability of a new node forming an edge with a node of large degree (Benevenuto et al., 2009). Finding triangles in graphs has many applications (Tsourakakis et al., 2009) (Chu & Cheng, 2011). The naïve approach requires generating all the possible combinations of nodes, and for each of the combinations test whether the nodes form a triangle or not. A triangle $\Delta = (V_\Delta, E_\Delta)$ of a graph $G = (V, E)$ is a three node subgraph with $V_\Delta = \{u, v, w\} \subseteq V$ and $E_\Delta = \{u, v\}, \{v, w\}, \{w, u\} \subseteq E$. In this section we study the triangle completion problem i.e., finding $(v_i, v_j) \notin$ E, such that if $\exists(v_i, v_j)$, $\Delta_f - \Delta_c$ is maximized, where Δ_f and and Δ_c are the counts of number of triangles in $G_m = (V, E \cup (v_i, v_j))$ and G, respectively

Social networks represent interaction among entities; groups can exist where entities are connected to one another directly or via others. "Close" groups exist where all the entities involved are connected to each other directly. Identification and creation of "close" groups have significant bearing in many real life scenarios: improve relations, expand business, influence political views etc. Therefore, studying the triangle completion problem is relevant. Let us consider the sample partial OSN graph shown in Fig. 5.1. In this case, connecting 'E' and 'G' results in 2 additional triangles, whereas connection 'F' and 'G' results in the maximum increase in the number of triangles by 6. Hence, we need an algorithm to find out the edge $e = (F, G)$ for the example graph shown in Fig. 5.1.

To solve this problem, we first need to find the number of triangles in the given graph G. Then we can proceed by choosing different edges that do not belong to G, and recalculate the total number of triangles in the modified graph G_m. Therefore, we need an algorithm for counting the number of triangles in the graph. As discussed

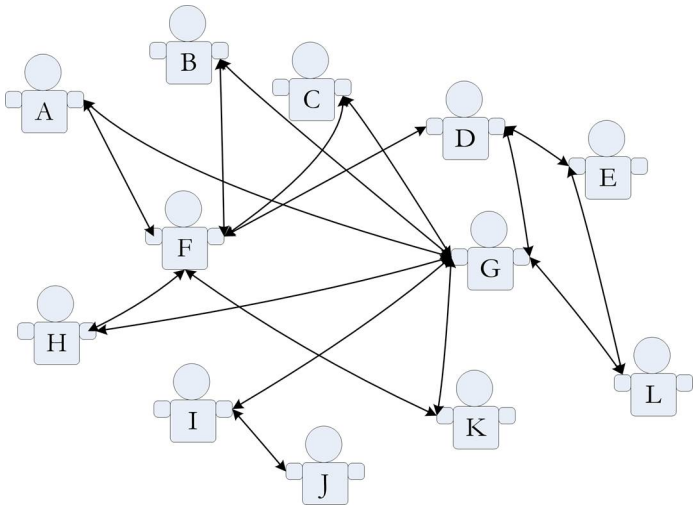


Figure 5.1: Triangle Completion Problem in OSNs

in our previous work (Chatterjee et al., 2013a), there are different techniques to do this. The brute-force method to do this generates all possible combinations of nodes, and tests for each of the combinations whether they form a triangle or not. This however tests for a lot of combinations that have no chance of being part of triangles. To reduce the search space, a breadth-first search (BFS) tree T of the graph G can be used. Now, according to the properties of BFS tree, nodes that form triangles must be in adjacent levels of T. Algorithm 5, described in Chapter 4 counts the number of triangles in graph G using T (Chatterjee et al., 2013a).

Now, the problem is to find an edge whose addition would increase the number of triangles by the largest number. This can be done by taking one of the following approaches:

I. Brute-force method: In this approach combinations are generated using two nodes that form the edge, and one other node from the remaining nodes. This would require testing for another ${}^nC_2$ combinations, where n is the total number of nodes in G.

II. Reconstructing BFS tree and recalculating: Since using BFS tree information reduces the search space, another approach would be to reconstruct a BFS tree for

the graph with the added edge. Using Algorithm 5 on the modified graph G_m, the number of triangles can be found. Now, this needs to be done for all possible edges that can be added to G.

III. Using previous BFS tree information: The previous approach requires reconstructing the BFS tree for each possible edge, and is computationally expensive. Therefore, an algorithm which makes use of the BFS tree that has already been generated before, would perform better; this leads to the following approach. When using the BFS tree data for the original graph, and adding a new edge, there can be the following four cases:

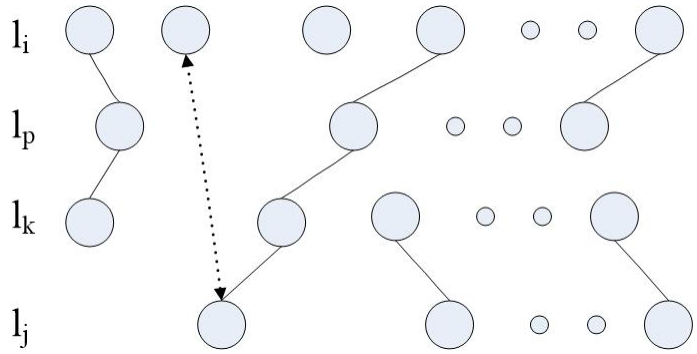


Figure 5.2: Partial BFS tree with edge connecting nodes over 4 levels

i. Edge connecting nodes separated by more than 2 levels: In the first case we consider all edges that connect nodes that are not within 2 levels adjacent to the current level in the BFS tree i.e., $\{v_i, v_j\} \notin E$ such that $v_i \in l_i$, $v_j \in l_j$, $\{l_i, l_j\} \in \{L\}$ and $|l_i - l_j| > 2$, as shown in Fig. 5.2. By the addition of edge $\{v_i, v_j\}$ the graph would be modified, with a single edge connecting the levels l_i and l_j. However, none of the other nodes in levels l_i and l_j have an existing edge; if that had been the case, then the nodes would not have been in these levels in the original BFS tree. Additionally, none of the nodes in levels l_k and l_p, where $|l_i - l_k| > 1$, $|l_j - l_p| > 1$, $|l_i - l_p| = 1$ and $|l_j - l_k| = 1$, can have an existing edge with nodes in both l_i and l_j which is required to form a triangle. Therefore,

all combinations containing nodes belonging to such levels can be excluded from calculations.

ii. Edge connecting nodes separated by 2 levels: In the second case we consider all edges that connect nodes belonging to levels in the BFS tree that are not adjacent i.e., $\{v_i, v_j\} \notin E$ such that $v_i \in l_i$, $v_j \in l_j$, $\{l_i, l_j\} \in \{L\}$ and $|l_i - l_j| = 2$, as shown in Fig. 5.3.

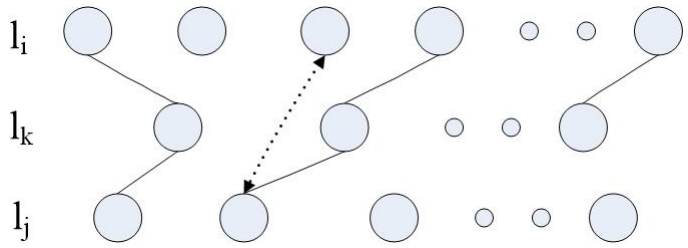


Figure 5.3: Partial BFS tree showing addition of an edge connecting nodes over 3 levels

In this case, the third node can belong to either of the levels l_i, l_j or l_k. However, if the third node belongs to levels l_i or l_j, then that combination would not form a triangle, since there would be only one edge between the levels l_i and l_j. Hence, the only case that has a potential to increase the number of triangles is when the third node belongs to the level in between i.e., l_k. So, the maximum increase in the number of triangles for this case is no more than the number of nodes in l_k.

iii. Edge connecting nodes in adjacent levels: In this case, the set of adjacent levels that contains the newly added node is the only one that needs to be recalculated. One approach would be to generate all possible combinations of nodes from within the set of levels, and count the number of triangles. The total number of combinations that need to be tested can be reduced by using some properties of the BFS tree. Let $\{v_i, v_j\}$ be the edge that is added, and $v_i \in l_i$, $v_j \in l_j$, and $l_i < l_j$. Now, if $v_i > v_k$, i.e., the node numbering is greater, where v_k is the parent of v_j in the original BFS tree, then any node that has a lower numbering than v_k in

the same level cannot be part of the combination for testing.

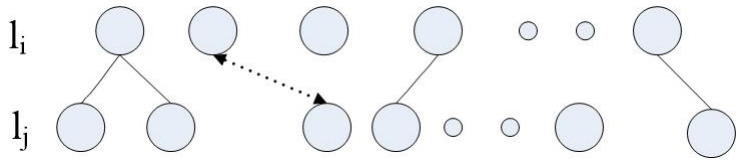


Figure 5.4: Partial BFS tree with edge connecting nodes over 2 levels

iv. Edge connecting nodes in same level: In this case, the third node comes from either the same level, or from the adjacent level in the set of the BFS tree. Therefore, the maximum number of triangles that can be formed in this case is equal to the number of nodes in the level of the parent plus the number of nodes in the level of the nodes forming the edge minus two, for the two nodes. Algorithm 6, implements the different possibilities as discussed above.

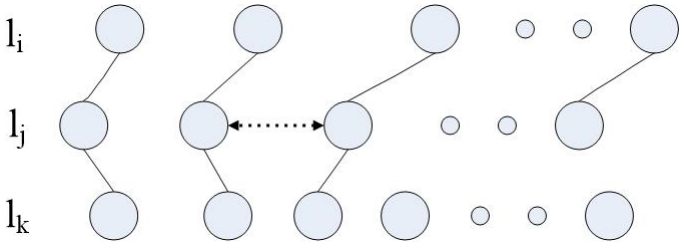


Figure 5.5: Partial BFS tree with edge connecting nodes in same level

5.4 Deleting edges in graphs

The structure of graphs can change by addition as well as deletion of edges. Deletion of edges can result from the elimination of a specific edge or a node. In the case of elimination of a specific edge, the nodes remaining as before i.e., $G = (V, E)$ is modified to $G_m = (V, E \setminus (v_i, v_j))$. Alternatively edges can be deleted by removing one end node of a specific edge; this might result in deletion of additional edges that are connected to it. The modified graph $G_m = (V \backslash v_i, E \backslash \Sigma(v_i, v_j))$ would contain one node less along with all edges associated with that node deleted too. Fig. 5.6 shows a

Algorithm 6: Identifying edge for maximum increase in number of triangles in G without recalculating BFS tree

```
Input: Graph G = (V, E)
Output: Edge {v_i, v_j} such that {v_i, v_j} ∉ E
begin
    T ← generateBFSTree(G);
    {L} ← splitIntoLevels(T);
    triangleCount ← 0;
    foreach {v_i, v_j} ∉ E do
        thisCount ← 0;
        if v_i ∈ l_i, v_j ∈ l_j, {l_i, l_j} ∈ {L} && |l_i − l_j| > 2 then
            thisCount ← 0;
        else if v_i ∈ l_i, v_j ∈ l_j, {l_i, l_j} ∈ {L} && |l_i − l_j| > 1 then
            forall the {v_k} ∈ l_k, where l_i < l_k < l_j do
                if isTriangle {v_i, v_k, v_j} then
                    thisCount++;
                end
            end
        else if v_i ∈ l_i, v_j ∈ l_j, {l_i, l_j} ∈ {L} && |l_i − l_j| > 0 then
            forall the {v_k} ∈ {l_i ∪ l_j \ v_i, v_j} do
                if isTriangle {v_i, v_k, v_j} then
                    thisCount++;
                end
            end
        else
            forall the {v_k} ∈ {l_k ∪ l_i \ v_i, v_j}, where l_i < l_j < l_k do
                if isTriangle {v_i, v_k, v_j} then
                    thisCount++;
                end
            end
        end
        if thisCount > triangleCount then
            triangleCount ← thisCount;
            edge ← {v_i, v_j};
        end
    end
    Output ← edge;
end
```

sample graph, where the number of components are increased by a maximum number by the deletion of a single node.

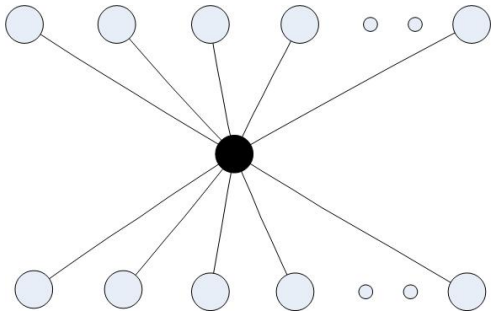


Figure 5.6: Graph depicting nodes with potential of maximum increase in the number of components with deleting of a single node

In this section, we focus on the effect of deletion of edges and the change in the number of triangles and connected components in the graph. Studying the effect of edge deletion has many practical applications. In case of road networks, identifying edges i.e., road segments whose removal would result in places being inaccessible is important. Also, creation of new road segments that help reduce the distance between junctions is of practical significance. Consider the road network as shown in Fig. 5.7.

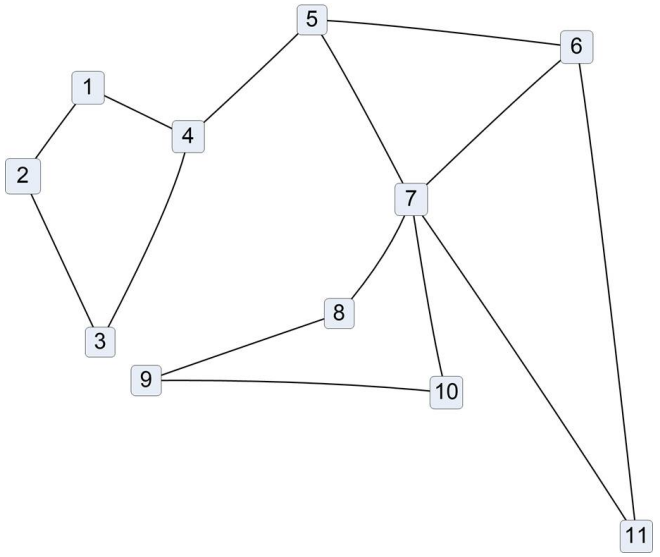


Figure 5.7: Graph showing sample road network and effect of edge deletion

The road segment $r = (4, 5)$ in Fig. 5.7, if removed would divide the graph into

two separate components. If this road segment or edge in the graph is removed, there would be no way to move between the two separate areas that are created. Similarly, if a new road segment is built to connect nodes 4 and 8, then it would reduce distance between junctions and also reduce the chance of portions of the graph becoming inaccessible because of removal of one road segment. Therefore, studying the problem of edge deletion is relevant.

5.4.1 Decrease in the number of triangles

In this sub-section we study an algorithm to identify an existing edge whose removal would increase the number of connected components in the graph. Specifically, given a graph $G = (V, E)$, find an edge $(v_i, v_j) \in$ E, such that $\Delta_c - \Delta_f$ is maximized, where Δ_f and Δ_c are the number of triangles in $G_m = (V, E \setminus (v_i, v_j))$ and $G = (V, E)$, respectively.

The approach to this problem is similar to finding an edge that results in the maximum increase in the number of triangles. However, since an existing edge is deleted in this case, considering only the adjacent levels of nodes in the BFS tree is sufficient as edges can exist only within that set. In Algorithm 7, the above procedure is described.

5.4.2 Increase in the number of connected components

In this sub-section, we focus on the effect of deletion of edges and the change in the number of connected components in the graph. It can be noted that the maximum increase in the number of components by deleting an edge is 1, as an edge can connect only 2 vertices which might otherwise be part of separate components.

Therefore, we want to find edges whose deletion would increase the number of components by 1. Finding the maximum increase in the number of components by the deletion of a node involves lot more calculations. Also, deleting a node might

increase the number of components by any number, the upper limit being dictated by the number of nodes. In particular, deleting a node might increase the number of connected components by α, where $0 \leq \alpha \leq n-2$.

Algorithm 7: Identifying edge whose removal results in maximum decrease in number of triangles in G

```
Input: Graph G = (V, E)
Output: Edge {v_i, v_j} such that {v_i, v_j} ∉ E
begin
    T ← generateBFSTree(G);
    {L} ← splitIntoLevels(T);
    triangleCount ← 0;
    foreach {v_i, v_j} ∈ E do
        thisCount ← 0;
        if v_i ∈ l_i, v_j ∈ l_j, {l_i, l_j} ∈ {L} && |l_i − l_j| > 0 then
            forall the {v_k} ∈ {l_i ∪ l_j \ v_i, v_j} do
                if isTriangle {v_i, v_k, v_j} then
                    thisCount++;
                end
            end
        else
            forall the {v_k} ∈ {l_k ∪ l_i \ v_i, v_j}, where |l_i − l_k| = 1 do
                if isTriangle {v_i, v_k, v_j} then
                    thisCount++;
                end
            end
        end
        if thisCount > triangleCount then
            triangleCount ← thisCount;
            edge ← {v_i, v_j};
        end
    end
    Output ← edge;
end
```

In the other case, the problem is to find the edges whose deletion would increase the number of connected components in the graph. Now, the brute-force approach would be to choose one edge at a time from the set of edges, delete the edge and then perform a breadth-first search and count the total number of components. Then

this number can be compared with the one found by performing the breadth-first search with all the edges in the original graph. This technique is similar to using the BFS tree properties for biconnected components decomposition (Eckstein, 1979). Although, this approach is simple, but it is computationally expensive because of the need to perform BFS on the graph once for each of the edges. An advanced approach would be to try to decrease the calculation of the entire BFS if certain criteria are met. This can be done by taking into account the information of the original BFS tree. Let an edge (v_i, v_j) be deleted from the graph, where $v_i \in l_i$ and $v_j \in l_j$. Now, in the original graph, in addition to the number of components in the entire graph, counts for the number of components in the subgraphs induced by the nodes in sets of adjacent levels and also those involving combinations of different sets of contiguous levels can be calculated and stored. Therefore, the number of components induced by the nodes and the modified number of edges for any specific set of adjacent levels are recalculated. If the number of connected components are identical, then it can be concluded that the deletion of the edge (v_i, v_j) in question does not change the number of connected components in the entire graph. However, if the number of connected components increases, then additional verification is required before any conclusion can be made about the changes. This can be done by taking into consideration two more levels, one the previous of the first level, and the other the next level of the second level. If the number of connected components still differ from the one calculated for the original graph, the process can be continued till there are no more previous or next levels left in the BFS tree of the original graph i.e., the entire graph has been considered and the BFS has been performed on it (we have the number of components for all combinations of contiguous levels calculated already to compare with). The above methodology can be improved by adding either the previous or the next level instead of both in a single step. By taking into account the densities of the edges within the adjacent levels, the one with the higher value among the previous

and next can be chosen.

Algorithm 8: Identifying edge $\{v_i, v_j\} \in E$ whose deletion leads to an increase in number of components in $G = (V, E)$

Input: Graph $G = (V, E)$
Output: Edge $\{v_i, v_j\} \in E$
begin
  $T \leftarrow$ `generateBFSTree(`G`)`;
  $\{L\} \leftarrow$ `divIntoConsecutiveLvlSets(`T`)`;
  $\{C\} \leftarrow$ `calComponents(`L`)`;
  componentCount $\leftarrow 0$;
  foreach $\{v_i, v_j\} \in E$ **do**
    $L_c \leftarrow L_i \setminus (v_i, v_j)$, where $(v_i, v_j) \in L_i$);
    count $\leftarrow$ `calComponents(`L_c`)`;
    while *(count* $> C_c$ *&&* L_{beg}&&$L_{end} \notin L_c$ **do**
      if $L_{beg} \in L_c$ **then**
        $\{L_c\} \leftarrow \{L_c\} \cup \{L_j\}$, where $\{L_c\} < \{L_j\}$;
        count $\leftarrow$ `calComponents(`L_c`)`;
      else if $L_{end} \in L_c$ **then**
        $\{L_c\} \leftarrow \{L_c\} \cup \{L_h\}$, where $\{L_h\} < \{L_c\}$;
        count $\leftarrow$ `calComponents(`L_c`)`;
      else
        $\{L_c\} \leftarrow \{L_c\} \cup \{L_h\} \cup \{L_j\}$, where $\{L_h\} < \{L_c\} < \{L_j\}$;
        count $\leftarrow$ `calComponents(`L_c`)`;
      end
    end
    if *count* $> C_c$ **then**
      edge $\leftarrow \{v_i, v_j\}$;
    end
  end
  Output $\leftarrow$ *edge*;
end

In Algorithm 8, the above procedure is described. First the breadth-first tree T is generated using the method *generateBFSTree*. Then T is divided into adjacent level sets using the method *divIntoConsecutiveLvlSets*. Thereafter, using *calComponents* the counts for the number of components for the various combinations of contiguous levels can be calculated and stored in C. In Algorithm 8, levels in T consisting the

edge in consideration are taken into account. Since nodes from previous and next levels are added to the data being computed on, care must be taken to not add levels beyond the first and last levels in T. Therefore, variables L_{beg} and L_{end} are used to store the first and last levels of T respectively. L_c consists of the nodes belonging to the levels that are part of the current computation. C_c denotes the count of the number of components that belongs to the current calculation. The *while loop* terminates if the count for components is equal to the one calculated before, or when both the first and the last levels of T are part of the nodes in consideration thereby indicating there are no more nodes to be added.

5.5 Real-world graph properties

For the purpose of analyzing graphs, we refer to real world data sets (Leskovec et al., 2009). The Stanford Large Network Dataset Collection framework provides a huge collection of real-world graphs (Stanford Network Analysis Project, 2011). The following graphs are considered in this chapter:

1. Texas Road Network (TRN): This graph represents the road network of Texas, where intersections and endpoints are represented by nodes, and the roads connecting the same are represented by undirected edges.

2. Pennsylvania Road Network (PRN): This is the graph depicting the road network of Pennsylvania; the representation is similar to that of the TRN.

3. California Road Network (CRN): The road network of California is depicted using this graph; the representation is similar to both TRN and PRN. All the road network graphs are undirected.

4. Enron Email Network (EEN): This is an email communication network within a dataset of size approximately half million. The nodes are email addresses, and

edges exist between the nodes if there exists a communication between them.

5. Internet Topology Graph (ITG): This graph consists of traceroutes run on Autonomous systems using Skitter. This consist of an undirected graph comprising of Internet data from various routers depicting a part of the Internet.

6. Facebook Social Circles (FSC): This graph consists of anonymized data representing 'circles' i.e., 'friend lists' from Facebook. This undirected graph consists of data collected from survey participants using a specific Facebook application.

In Table 5.1, the data for the different graphs in the form of total number of nodes and edges are presented along with the largest connected component. These graphs

Graph	Nodes	Edges	Nodes in largest connected component
TRN	1,379,917	3,843,320	1,351,137
PRN	1,088,092	3,083,796	1,087,562
CRN	1,965,206	5,533,214	1,957,027
EEN	36,692	367,662	33,696
ITG	1,696,415	11,095,298	1,694,616
FSC	4,039	176,468	4,039

Table 5.1: Real World Graphs

are required to be processed on the GPUs. Therefore, to make use of the faster shared memory on the GPUs, the size of the data required for the computations must fit into the desired level in the memory hierarchy. As discussed in the earlier Sections, making use of the BFS tree information is useful for reducing the size of the search space, thereby reducing the space for the required data. Table 5.2 shows the number of levels in the BFS tree for each of the graphs, along with the maximum number of nodes in a single level, in any set of 2 and 3 consecutive levels.

Graph	# of levels in BFS tree	Max. # of nodes in a single level	Max. # of nodes any 2 cons. levels	Max. # of nodes any 3 cons. levels
TRN	723	4,426	8,782	13,122
PRN	557	4,057	8,108	12,130
CRN	557	6,115	12,180	18,258
EEN	9	16,114	30,167	32,328
ITG	22	748,276	1,299,877	1,623,942
FSC	7	1,742	2,913	3,432

Table 5.2: Real World Graphs: BFS Tree Level Information

Although the number of nodes in the different sets of levels in the BFS tree reduces the number of nodes for which adjacency data is required, it might still be more than what is necessary. It can be noticed that the subgraph induced by the nodes in each of these sets of levels might form more than a single connected component. Nodes in separate connected components are not required to be tested for the properties that are being studied, and hence are not required to be stored together. In Table 5.3 we provide information about the number of connected components and the largest connected component while considering sets of 2 and 3 levels from the BFS tree.

Graph	# of conn. comp. 2 levels	Nodes in largest connected component 2 levels	# of conn. comp. 3 levels	Nodes in largest connected component 3 levels
TRN	2,033	73	1,558	247
PRN	1,887	136	1,446	558
CRN	3,147	92	2,545	195
EEN	4,455	24,512	4,454	26,610
ITG	10,539	1,287,183	10,340	1,611,636
FSC	13	2,900	13	3,419

Table 5.3: Real World Graphs: Connected Components in BFS Tree Levels

For studying the properties of the graphs, preprocessing is done on the graph data using the CPU. Reading the data from edge lists and storing it in appropriate data structures is followed by performing and creating the BFS tree of the same. The overhead incurred in doing so is reported for the real-world graphs being considered in Table 5.4.

Graph	Reading edge list into CPU data Structure (seconds)	Generating BFS tree (seconds)
TRN	1.39	0.15
PRN	1.08	0.12
CRN	1.98	0.23
EEN	0.17	0.01
ITG	6.89	0.29
FSC	0.03	0.01

Table 5.4: Overhead for reading data from edge list and generating BFS tree

The above analysis provide some valuable insight. Considering the structure of the graph and using breadth-first search tree information, the number of nodes in components spanning across levels are much smaller compared to the actual size of the graph. This leads to the next section, where this additional information is taken into account, and a heuristic is provided for approximate calculations.

5.6 Approximate counting

For large graphs, performing exact calculations is computationally expensive. To find an edge whose addition increases the count of triangles by the maximum number requires checking for all the edges that are not part of the original graph in the naive case. For the graphs considered in this chapter, with over a million nodes, that would require huge amount of calculations. In the problem of counting triangles and changes

to the number by addition or deletion of edges, data only from adjacent levels of BFS tree are required. However, since all the levels would have to be considered, the total combinations considered is still significant. A better approach would be to test part of the graph and provide a result within an acceptable percentage of error. We consider a heuristic, where we choose the edge to be added to be present only within the set of maximum edge density. The heuristic is given in the form of the Algorithm 9.

Algorithm 9: Heuristic for approximation in identifying edge $\{v_i, v_j\} \notin E$ for maximum increase in number of triangles in $G = (V, E)$

```
Input: Graph G = (V, E)
Output: Edge {v_i, v_j} such that {v_i, v_j} ∉ E
begin
    T ← generateBFSTree(G);
    {L} ← divIntoConsecutiveLvlSets(T);
    {L_i} ← maximumDensity(L);
    triangleCount ← 0;
    foreach {v_i, v_j} ∉ E && v_i ∈ L_i && v_j ∈ L_i do
        G_m = (V_{L_i}, E_{L_i} ∪ (v_i, v_j));
        T_m ← generateBFSTree(G_m);
        thisCount ← Algorithm5(T_m);
        if thisCount > triangleCount then
            triangleCount ← thisCount;
            edge ← {v_i, v_j};
        end
    end
    Output ← edge;
end
```

However, Algorithm 9 still considers all the nodes in the level with the maximum density. The heuristic can be improved by considering only the largest connected component within the level of the BFS-tree in consideration. Using a function *find-LargestComponentLevel*, the largest connected component C_i in the specific level can be found out. Then, all edges that do not exist in the original graph which can be formed using the nodes in C_i are considered one by one by adding to the graph induced by the nodes in C_i and checked for triangles using Algorithm 5. Using this

technique Algorithm 9 can be improved.

5.7 Experimental results

Various approaches for solving different graph problems have been discussed in this chapter. The basic target is to reduce the solution space, thereby decreasing the number of computations and resulting in faster outputs. Table 5.5 shows the number of computations required for solving the triangle completion problem on the real-world data sets using naive computation, breadth-first search tree information with all levels, with adjacent levels containing maximum number of nodes and heuristic considering the largest connected component within the levels of BFS-tree under consideration.

Graph	Naive Computation	Using BFS Tree Info.	Using Adj. levels Max. nodes	Using Heuristic Largest Comp.
TRN	4.37×10^{17}	1.30×10^{13}	1.12×10^{11}	6.21×10^{4}
PRN	2.14×10^{17}	9.20×10^{12}	8.88×10^{10}	3.74×10^{5}
CRN	1.26×10^{18}	4.97×10^{13}	3.01×10^{11}	3.42×10^{4}
EEN	8.23×10^{12}	4.96×10^{12}	4.57×10^{12}	2.45×10^{12}
ITG	8.13×10^{17}	5.06×10^{17}	3.66×10^{17}	3.55×10^{17}
FSC	1.09×10^{10}	5.49×10^{9}	4.11×10^{9}	4.06×10^{9}

Table 5.5: Comparison for number of computations

Fig. 5.8 shows the comparison between the number of computations for the various approaches as reported in Table 5.5; the different graphs under consideration are shown along the X-axis and the number of computations are plotted along the Y-axis using a log-scale with the base 2. It is evident, the number of computations are

reduced greatly using the advanced approaches. Since the number of computations required for the naive approach is huge, it is not depicted in the graph. It can be noted that the data plots representing the Enron Email Network (EEN) and Facebook Social Circles (FSC) show minor improvement between the different cases due to the large number of nodes being present in the level of the BFS-tree containing the maximum number of nodes.

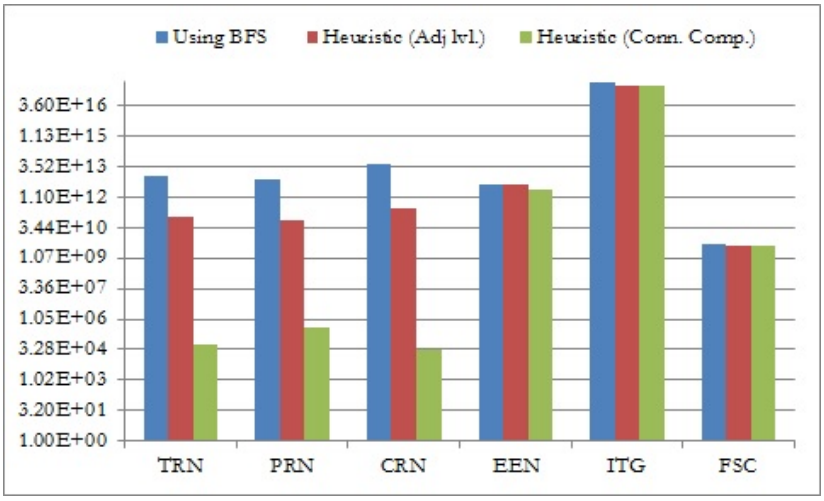


Figure 5.8: Comparing the number of computations performed on the data using the various approaches

The triangle completion problem described in this chapter is implemented using both CPU and GPU. The CPU consists of quad-core 2.27 GHz Intel Xeon processors and the GPU used for the experiments is Nvidia Tesla K20m. The CPU implementation uses a single thread.

Fig. 5.9 shows the timings for executing the programs on the CPU and GPU; the different graphs data are shown along the X-axis and the time required for the computations in seconds are plotted along the Y-axis using a log-scale with the base 2. As evident from the plots, the adjacency level information from the BFS-tree helps improve the performance of the calculations. With the help of a significant reduction in the solution space, the heuristic performs better than the other approaches. The results show a 10 $\times$ speedup on the GPU using the adjacency level information of

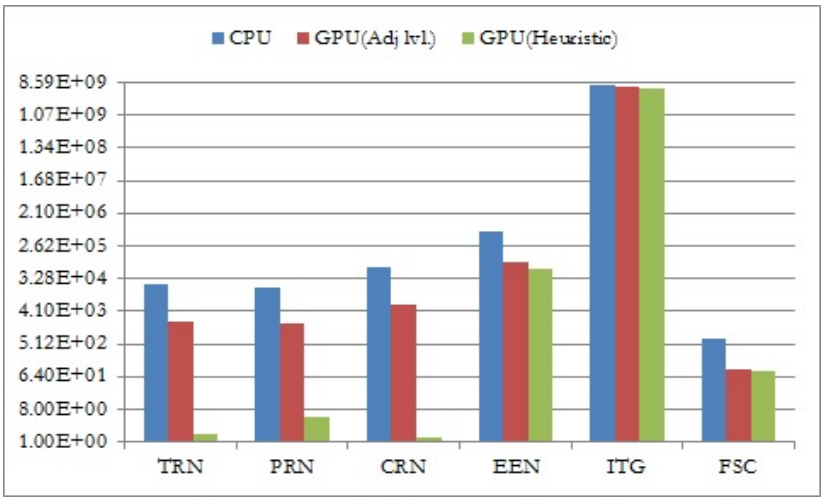


Figure 5.9: Comparing timings for larger graphs

the BFS tree as compared to a sequential implementation for majority of the data sets considered; the performance is even better for approximate calculations when considering the proposed heuristic. The error introduced due to the proposed heuristic is less than 6%. The only cases where the speedup achieved was less were contributed by the factor of the sizes of the data, that could not be divided to be processed in parallel, using the level information of the breadth-first search tree.

5.8 Summary

In this chapter we study the properties of large graphs, specifically Online Social Networks and Road Networks. We analyze the changes in the graphs considering the dynamic nature of the same by taking into account the effects of adding edges into and deleting edges from the graph. We provide an algorithm for triangle completion, where we find an edge in a given graph whose addition would result in the maximum increase in the number of triangles in the same. Similarly, we provide an algorithm to study the effect of deleting an edge from a graph resulting in the maximum decrease in the number of triangles in the graph and also an increase in the number of connected components in the same. The algorithms take into consideration the structure of the

graph primarily based on the breadth-first search tree information. These approaches not only help reduce the number of calculations but also eliminate unnecessary data thereby helping solve the problems by storing the data in smaller but faster levels in the memory hierarchy of the GPUs. Since finding exact solutions requires a huge amount of computation, a heuristic is introduced to find approximate solutions and the methodology is discussed and compared. This chapter has been adapted from one of our research papers (Chatterjee et al., 2014).

Chapter 6

Graph Compression

Graphs can be stored in the memory in many different ways using various data structures. The memory on the GPU i.e., the device is limited; specifically, the faster levels of memory in the hierarchy, say the shared memory, have less space than the ones with higher memory access latency, say the global memory. To perform analysis on graphs, data required for computation must be stored on the device memory. The memory transfer overhead can be reduced if more data can be copied and stored in the device memory. Therefore, studying techniques to store data efficiently is important. In this chapter we discuss graph compression techniques which can address the above mentioned issues.

6.1 Introduction

Graphs can be stored using a variety of data structures. The most common data structures that are used to represent graphs are adjacency matrix and adjacency list. The choice of the data structure also depends on the characteristic of the graph data. If the graph is dense, then it might be useful to store the data in an adjacency matrix. However, if the graph is sparse, and most of the elements are 0 values, then it wastes a lot of space by using the adjacency matrix, and in this case using an adjacency list might be an efficient choice. When the graph data is large or the memory used to store the graph has limited space, then methods must be employed to reduce the size required to store the graph in order to perform computations on the same. Therefore, techniques must be designed to compress the relevant data and store it in

the provided memory. Some techniques require decompressing the data for it to be retrieved. However, such techniques would not be suitable if the memory is limited. Therefore, it is important to be able to perform computations on the compressed data itself. However, the increased complexity in retrieving the data from the compressed representation can also be a factor in the choice of the compression technique.

6.2 Graph compression techniques

As with any other compression techniques, graph compression techniques can also be sub-divided into two broad categories: a) Lossy compression b) Lossless compression. In the case of lossy compression techniques, the original data cannot be exactly retrieved after decompressing the compressed data; whereas, in the case of lossless compression techniques, there is no loss of information, and the original can be retrieved in entirety from the compressed data. In this chapter we are interested in studying only lossless compression techniques.

6.2.1 Overview of techniques

In general, most graph compression techniques can be broadly divided into the following categories (Deshpande, A., 2010).

- **Replacing specific structures** This approach involves identifying and replacing specific structures like a clique, with a special node and edges to the nodes in the clique.
- **Adjacency information similarity** Redundant data in the form of similar neighbors for nodes can be replaced by storing the data once and using pointers to the data for the rest.

- **Modify graph layout** The basic idea is to linearize the nodes so that average "stretch" of an edge is minimized. This is also referred to as "minimum-linear-arrangement" problem.

6.2.2 Related work

Graph compression techniques have been studied for a long time resulting in a number of methods being developed. Identifying and replacing a specific structure with an equivalent efficient one is one such technique. The m×n edges in a complete bipartite graph $K_{m,n}$ can be replaced using a special node and m+n edges (Feder & Motwani, 1991) (Buehrer & Chellapilla, 2008).

The Web can be modeled as a graph where the addresses are the nodes and hyperlinks between the same are the edges. Compressing adjacency lists using pointers to other similar lists and storing data for additions and deletions achieves high compression (Randall et al., 2002).

Using common neighborhoods and exploiting locality information can be used to compress graphs with power law distribution (Boldi & Vigna, 2004).

Lexicographic ordering along with neighborhood information is effective in compressing graphs (Chierichetti et al., 2009); however, this mechanism does not work for social network graphs which do not have any natural order.

6.3 Quadtree representation

Quadtree is a data structure which is used to normally represent images using partitioning of the two dimensional space by recursively subdividing into four quadrants or regions; each internal node of the quadtree has exactly four children (Samet, 1985). The most common type of quadtree is the Point-Region Quadtree, also referred to as the PR quadtree. A PR quadtree represents data points in a two dimensional region.

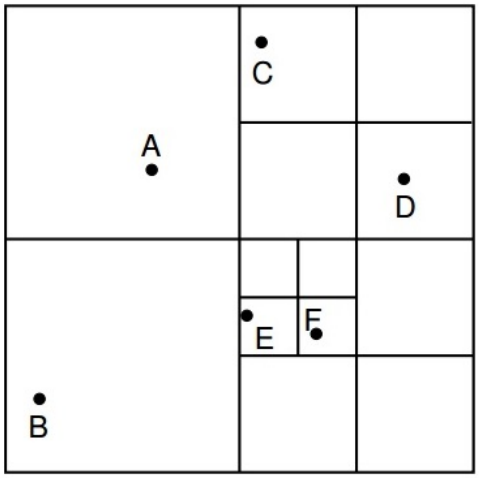


Figure 6.1: A map of data points for a two dimensional region

The region is subdivided into four quadrants if it contains more than 1 point in it. If a region contains a single point or no points, it is designated by a leaf node in the representation. A sample region data is shown in Fig. 6.1 and the corresponding PR quadtree is shown in Fig. 6.2.

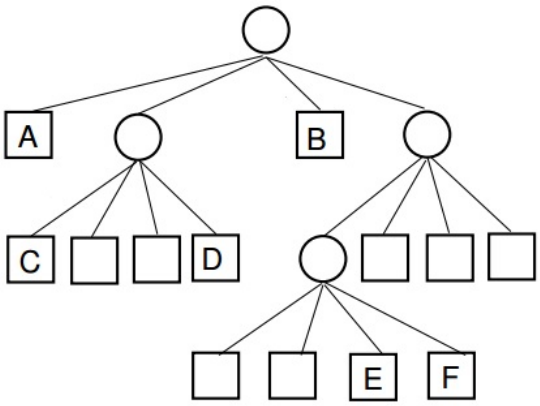


Figure 6.2: The PR Quadtree for the region shown in Fig. 6.1

6.3.1 Graphs as Quadtree

Quadtree representation can also be used to store graphs efficiently. The quadrants of an adjacency matrix of a graph can be converted and stored according to the row major order. The decision to further expand a quadrant into four sub-quadrants is taken based on the data present in the same. If the data matches with one of the preselected values, then the quadrant is represented as a leaf node in the tree. Quadrants matching preselected values can be stored internally using integer values that map to the specific pattern. Therefore, using the quadtree representation, the entire graph information can be stored in the form of an array using bits. The contents of the bit array can be stored as follows:

- **0**: all 0's in quadrant
- **1**: all 1's in quadrant
- **2**: 0's in diagonal, and rest 1's
- **3**: the quadrant needs to be expanded further

Since there are only 4 types of values, using 2 bits to represent each quadrant is enough when storing in the bit array. Consider the following graph shown in Fig. 6.3.

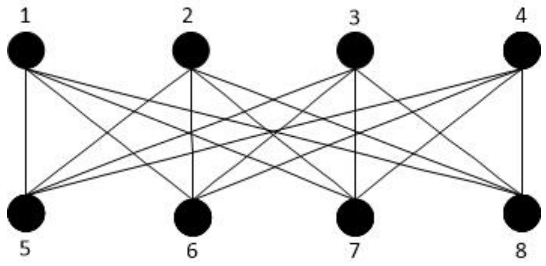


Figure 6.3: A sample graph

To convert to a quadtree, the adjacency matrix of the graph is considered. The adjacency matrix for the graph in Fig. 6.3 is given in Fig. 6.4.

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0

Figure 6.4: Adjacency matrix of graph shown in Fig. 6.3

The quadtree representation of the graph above is shown as follows:

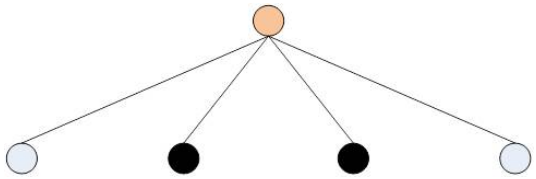


Figure 6.5: Quadtree representation of graph shown in Fig. 6.3

The byte representation of the quadtree is given by $Q = \{3, 0, 1, 1, 0\}$. The value 3 corresponds to the quadrant covering the entire adjacency matrix. The following values of 0, 1, 1 and 0 represents the top-left, top-right, bottom-left and bottom-right quadrants respectively.

Algorithm 10 describes the method for generating the quadtree from the adjacency matrix of a graph. The method *CheckMatrixUniformity* checks whether the adjacency matrix provided as the parameter is uniform or not. Depending on the data, the method returns a 0, 1, 2 or 3 for all 0's, all 1's, 0's in diagonal and rest 1's, and non-uniform matrix respectively. For the first 3 cases, the returned value is added to the quadtree representation; for the last case Algorithm 10 is called recursively for all the quadrants of the matrix generated by using the method *DivideMatrixIntoQuadrants*. For a graph $G = (V, E)$, where $|V| = n$, the algorithm loops over total of n^2 elements in each level of the quadtree. Since in the worst case there are $\log_2 n$ levels, the time

Algorithm 10: QuadGen: Quadtree generation from adjacency matrix

```
Input: Adjacency matrix A[ ][ ] of graph G
Output: Quadtree Q of G
begin
    Code ← CheckMatrixUniformity(A);
    if Code = 0 then
        Concatenate(Q, 0);
    else if Code = 1 then
        Concatenate(Q, 1);
    else if Code = 2 then
        Concatenate(Q, 2);
    else
        Quadrants{} ← DivideMatrixIntoQuadrants(A) ;
        forall the Quadrants_i ∈ Quadrants{} do
            QuadGen (Quadrants_i);
        end
    end
    Output ← Q;
end
```

complexity of the algorithm for generating the quadtree from the adjacency matrix is $O(n^2 \log_2 n)$.

6.3.2 Compression using quadtree

The sample graph shown in Fig. 6.3 consists of 8 nodes. Therefore, the space required to store the graph using the adjacency matrix representation is $8 \times 8 = 64$ bits. Now, the corresponding quadtree representation shown in Fig. 6.5 when stored in the bit array requires information for 5 elements, each of which requires 2 bits of data. Therefore, the space required to store the quadtree representation is $5 \times 2 = 10$ bits. Hence, in this case, the graph data is compressed using the quadtree representation, and requires just $\frac{1}{6}$ of the original space.

6.3.3 Numbering matters

Let us consider the graph shown in Fig. 6.6 and its corresponding adjacency matrix and quadtree representation. It can be noted that the graph structure is similar to the one shown in Fig. 6.3.

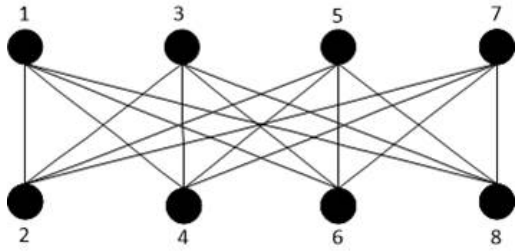


Figure 6.6: Sample graph with nodes numbered in a specific way

0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

Figure 6.7: Adjacency matrix for the sample graph shown in Fig. 6.6

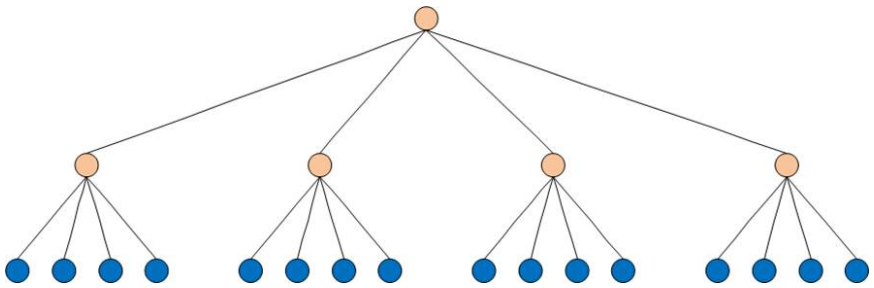


Figure 6.8: Quadtree representation for the sample graph shown in Fig. 6.6

Since the quadtree has 21 elements, where each element requires 2 bits, the memory required to store the graph is 42 bits. In this case, while the structure of the graph

shown is similar to the one before, the numbering of the nodes is changed. In the case of the renumbered graph, since the adjacency matrix changed, the corresponding quadtree also changed resulting in 21 elements compared to just 5 elements that required 10 bits for storage. The size required for representing the graphs is directly proportional to the number of quadrants that are non-uniform. Since the adjacency matrix varies with the numbering of the nodes, some combinations might be better than others. Therefore, renumbering nodes to make quadrants of the matrix uniform is an important step in using quadtrees for compact representation of graphs.

6.3.4 Special graphs

In this sub-section we discuss how graphs with some specific properties can benefit from being represented using quadtrees. Following are the the special graphs that we study: a) Complete bipartite graph, b) Complete k-partite graph, c) Block graphs and d) Chordal graphs. We look into the details of each of the above mentioned types of graphs below:

1. **Complete bipartite graphs**: A graph $G = (V, E)$ is called bipartite if its vertex set can be partitioned into two disjoint subsets $V = V_1 \cup V_2$, such that every edge has the form $e = (a, b)$ where $a \in V_1$ and $b \in V_2$. A complete bipartite graph $K_{m,n}$ is a bipartite graph that has each vertex from one set adjacent to each vertex of another set.

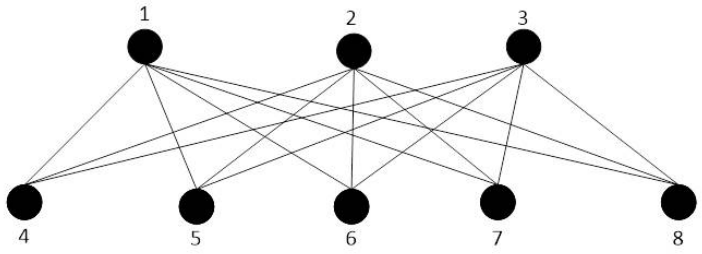


Figure 6.9: Sample complete bipartite graph

0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
1	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	0	0	0	0	0

Figure 6.10: Adjacency matrix for the sample complete bipartite graph

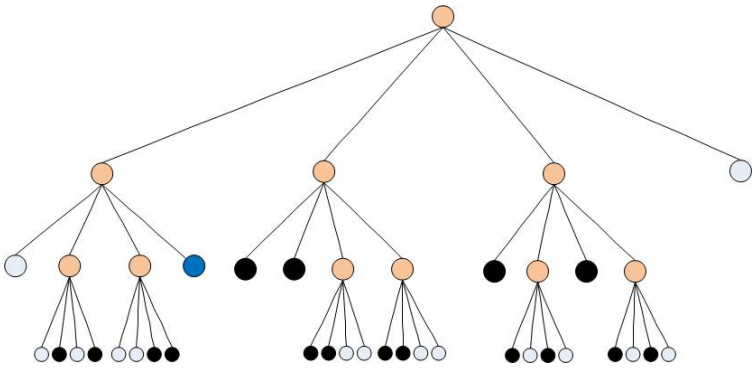


Figure 6.11: Quadtree representation for the sample complete bipartite graph

For the complete bipartite graph shown in Fig. 6.9, the adjacency matrix requires 64 bits while the quadtree representation requires 82 bits. While in this specific example the space required is more, in other cases it is a good candidate for representing using quadtrees, as can be seen from the sample graph shown before in Fig. 6.3. It has already been shown that in the previous example the quadtree requires just 10 bits instead of 64.

2. **Complete k-partite graphs**: A graph $G = (V, E)$ is k-partite if the vertices can be decomposed into k disjoint sets such that no two graph vertices within the same set are adjacent. A complete k-partite graph is one where every pair of graph vertices in the k sets are adjacent. If there are p, q, $\cdots$, r graph vertices

in the k sets, the complete k-partite graph is denoted $K_{p,q,\cdots,r}$. Fig. 6.12 shows $K_{3,2,3}$.

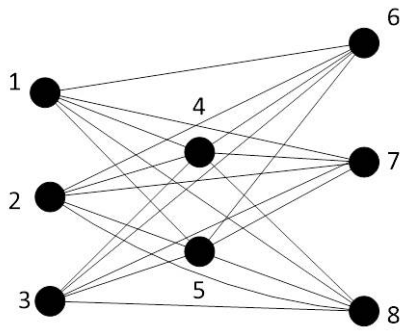


Figure 6.12: Sample complete k-partite graph

The adjacency matrix for the above graph is shown in Fig. 6.13.

0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
1	1	1	0	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0
1	1	1	1	1	0	0	0

Figure 6.13: Sample complete k-partite graph adjacency matrix

The quadtree representation for the above complete k-partite graph shown in Fig. 6.13 is given in Fig. 6.14.

3. **Block graphs**: An undirected graph $G = (V, E)$ is block graph or clique tree if every biconnected component is a clique. A sample block graph is shown in the following figure:

   The block graph may be a good candidate for representation using quadtree. It can be noted that a clique can be part of more than one quadrant, and therefore

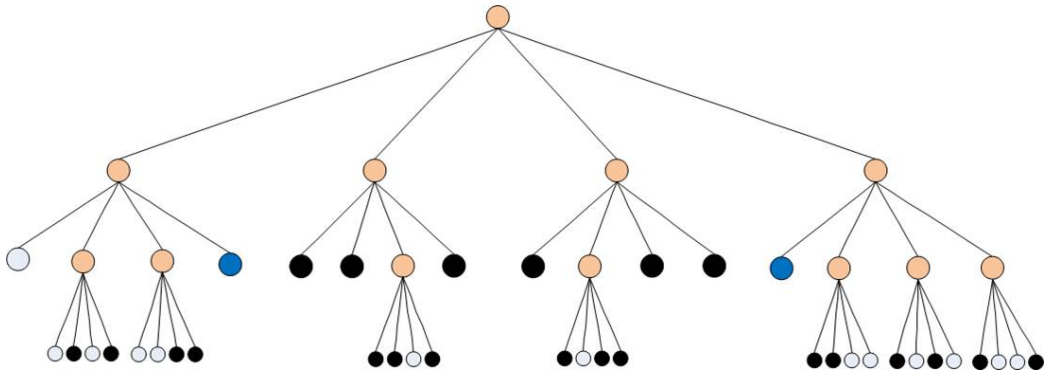


Figure 6.14: Quadtree representation for the sample complete k-partite graph

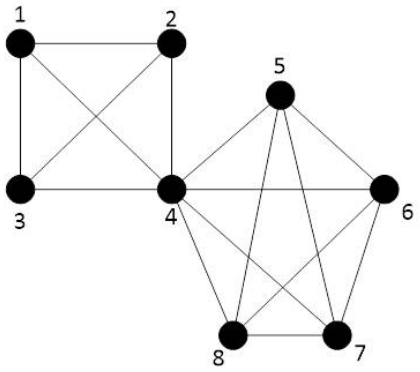


Figure 6.15: Sample block graph

can be subdivided into different quadrants. However, since a sub-clique is also a clique, the representation is still efficient.

The adjacency matrix and quadtree representation of the block graph shown in Fig. 6.15 is shown in Fig. 6.16 and Fig. 6.17 respectively.

The quadtree representation contains 29 elements and therefore requires 58 bits. So, in this case, the quadtree is actually efficient compared to the adjacency matrix representation which still requires 64 bits.

4. **Chordal graphs**: An undirected graph $G = (V, E)$ is chordal if every cycle of length greater than three has a chord i.e., an edge connecting two non-consecutive vertices on the cycle. Equivalently, every induced cycle in the graph should have at most three nodes. The chordal graphs may also be characterized

0	1	1	1	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	1	0	0	0	0
1	1	1	0	1	1	1	1
0	0	0	1	0	1	1	1
0	0	0	1	1	0	1	1
0	0	0	1	1	1	0	1
0	0	0	1	1	1	1	0

Figure 6.16: Sample block graph adjacency matrix

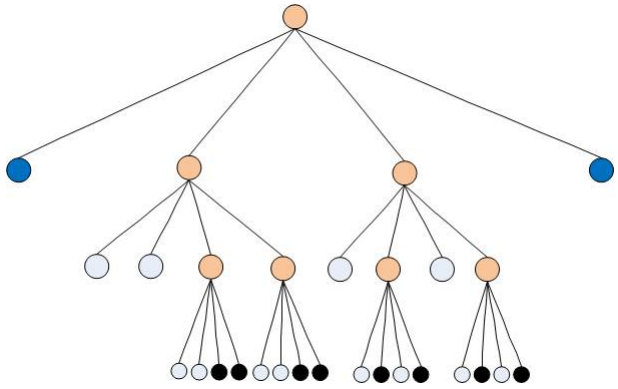


Figure 6.17: Quadtree representation for the sample block graph

as the graphs that have perfect elimination orderings (Chandran et al., 2003).

In a graph $G = (V, E)$, a vertex v is called simplicial if and only if the subgraph of G induced by the vertex set $\{v\} \cup N(v)$ is a complete graph, where $N(v)$ is the set of neighboring vertices of v. For example, in the graph shown in Fig. 6.18, vertex 3 is simplicial, while vertex 4 is not.

A graph G on n vertices is said to have a perfect elimination ordering if and only if there is an ordering $v_1, \cdots, v_n$ of G's vertices, such that each v_i is simplicial in the subgraph induced by the vertices $v_1, \cdots, v_i$. As an example, the graph shown in Fig. 6.18 has a perfect elimination ordering, witnessed by the sequence $(3, 1, 2, 4)$ of its vertices.

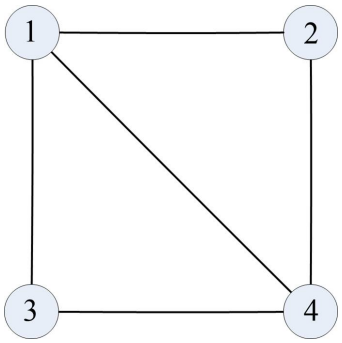


Figure 6.18: Sample graph showing a simplicial vertex

Fig. 6.19 shows a sample chordal graph.

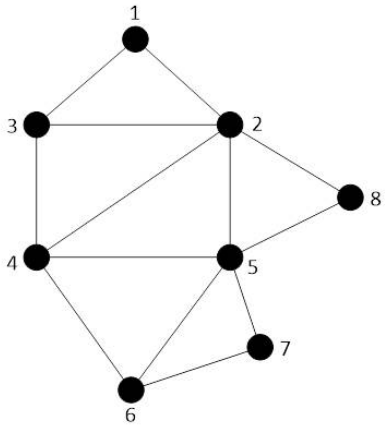


Figure 6.19: Sample chordal graph

Fig. 6.20 represents the adjacency matrix of the chordal graph shown in Fig. 6.19.

Fig. 6.21 depicts the quadtree representation of the chordal graph shown in Fig. 6.19.

Since there are 61 elements in the quadtree representation, the size required for the adjacency matrix representation is 64 bits and that of the quadtree is 122.

A Perfect Elimination Ordering (PEO) using the sample graph is given by the sequence $\{1, 3, 8, 7, 6, 2, 4, 5\}$. Renumbering the nodes according to the PEO, with the old numbers being shown in parentheses, the graph and its correspond-

0	1	1	0	0	0	0	0
1	0	1	1	1	0	0	1
1	1	0	1	0	0	0	0
0	1	1	0	1	1	0	0
0	1	0	1	0	1	1	1
0	0	0	1	1	0	1	0
0	0	0	0	1	1	0	0
0	1	0	0	1	0	0	0

Figure 6.20: Sample chordal graph adjacency matrix

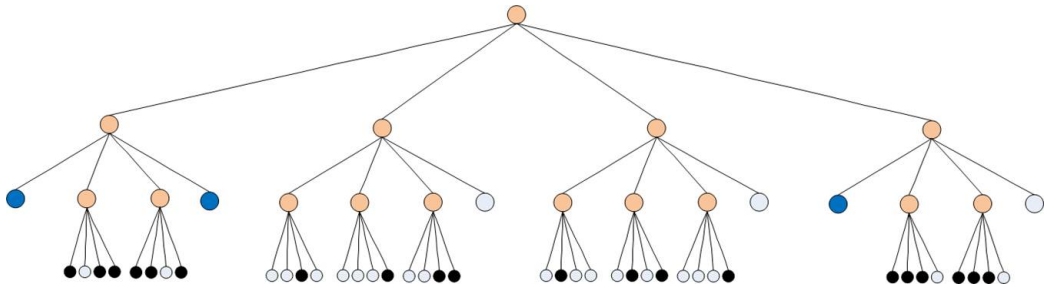


Figure 6.21: Quadtree representation for the sample chordal graph

ing quadtree representation is shown in Fig. 6.22 and Fig. 6.23 respectively.

The size of the quadtree for the renumbered graph is 90 bits. So, the size decreased by 32 bits by renumbering according to PEO, and this is an efficient technique for renumbering nodes.

6.3.5 Modifying graphs

As discussed before, the size of the quadtree is directly proportional to the number of non-uniform quadrants. Therefore, one way to compress graphs by representing using quadtree is to modify the graph by adding and deleting certain edges to make the adjacency matrix quadrants uniform. However, since the graph is modified, to retrieve the original data, information about the edges added and deleted must be stored too.

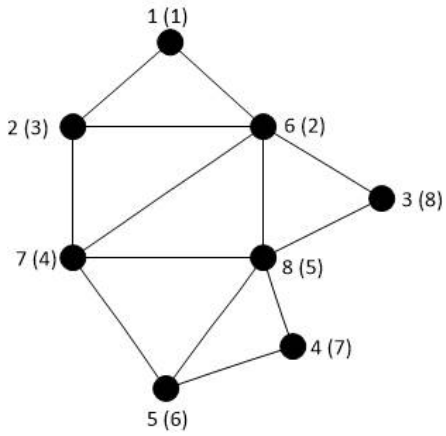


Figure 6.22: Sample chordal graph renumbered according to PEO

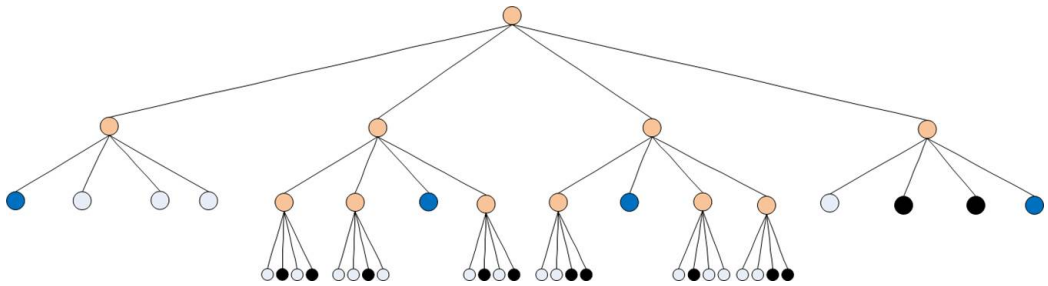


Figure 6.23: Quadtree representation for the sample renumbered chordal graph

Fig. 6.24 shows a modified version of the chordal graph given in Fig. 6.22. The corresponding quadtree is shown in Fig. 6.25. The size of the quadtree is 58 bits. Also, the information for the 3 edges need to be stored (2 removed and 1 added). $\log_2 n$ bits are required to store the nodes numbers for a graph with n nodes. Therefore, 6 bits are required to store each edge using an edge list representation; 3 bits for each of the end nodes for the graph containing 8 nodes. The extra space required to store this information is 18 bits. So, the total space needed is 76 bits; hence the space is reduced by another 14 bits compared to the PEO numbering of the chordal graph.

Further modifications can be made to the chordal graph to reduce space required. In addition to adding $(1, 8)$, $(4, 7)$ and removing $(4, 8)$, the following needs to change: add edge $(2, 5)$, remove edge $(2, 6)$. These changes reduce the quadtree size to 42 bits; an additional 30 bits are required to store the modified edge information. The total

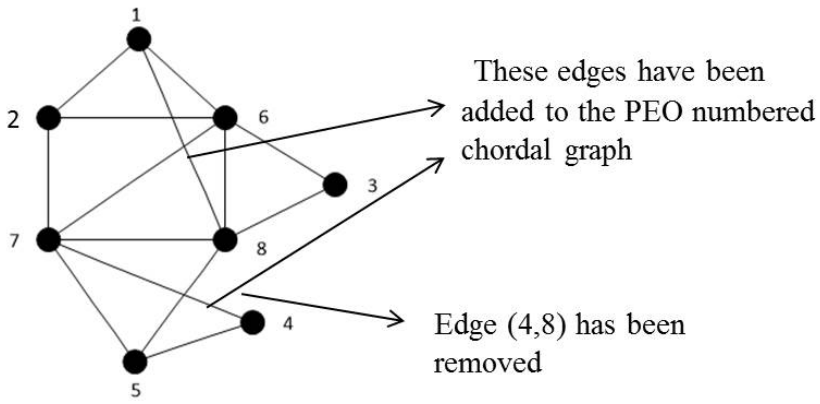


Figure 6.24: Modified chordal graph with edges added and removed

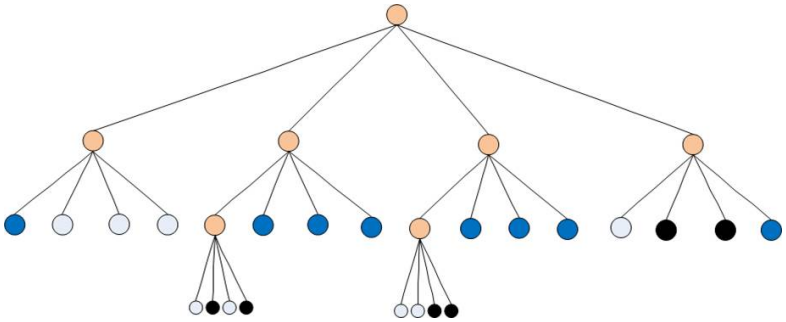


Figure 6.25: Quadtree representation of the modified chordal graph

size for this case is 72 bits, which is significantly reduced from the original size of 122 bits for the chordal graph.

6.3.6 Hybrid approach

It can be observed from the examples in the previous sub-sections that for many cases, the quadtree representation for graphs of size 8 require more than 64 bits, which is inefficient compared to the adjacency matrix representation. However, for larger graphs, the quadtree approach is efficient compared to other data structures. Even for larger graphs, when the quadrant reduces to 8×8 bits, the quadtree would require more space for further quadrant expansions. Therefore, a hybrid approach, where the recursive division of the quadrants stop whenever the quadrant size reaches 8×8 is a better technique. So, in the byte representation of the quadtree, an additional bit for

each element would be required to indicate whether the quadrant is further expanded or represented using adjacency matrix. Although this would need additional bits, overall the space required decreases, as verified by the experimental results.

6.3.7 Experimental results

The choice of data structure is influenced by the edge density of the graphs being considered. Therefore, it is important to analyze the space requirements for the graphs of different sizes with varying edge densities.

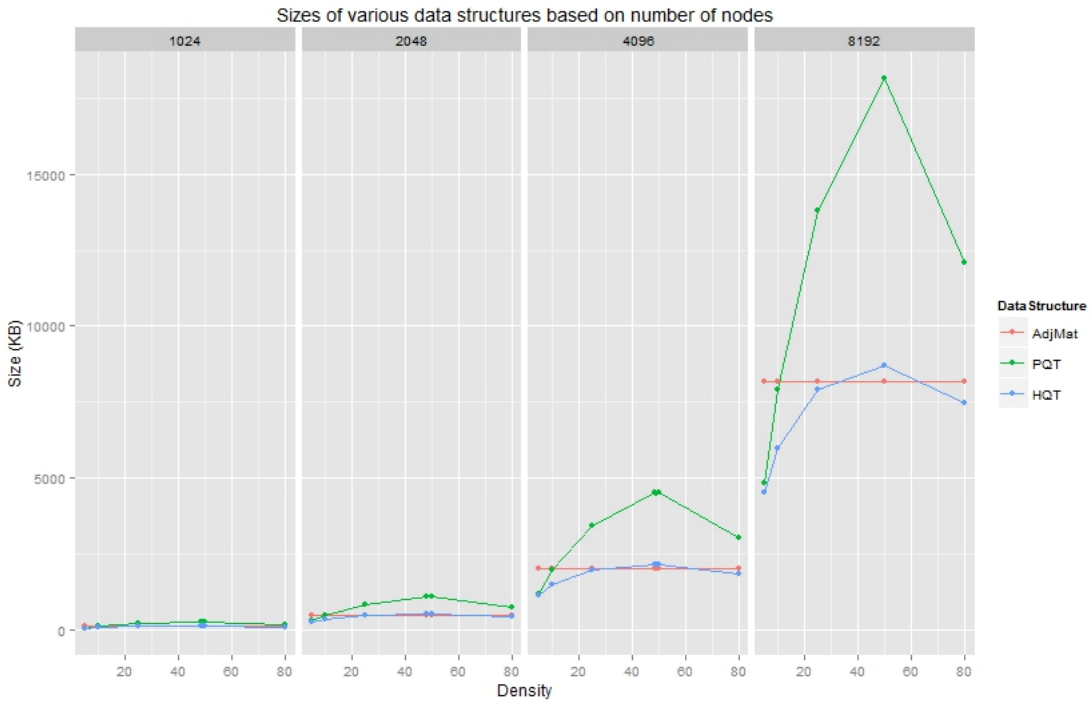


Figure 6.26: Data representation comparison for high densities

For the experiments, graphs of sizes 1024, 2048, 4096 and 8192 are considered. These sizes are considered because from the analysis of real-world data it can be seen that sizes of connected components considering level information from the BFS tree are within this range for most of the cases. The density of the graph is given by $\frac{e}{p} \times 100$, where e is the number of edges in the graph, and p is the total number of

edges possible. The densities that are considered in percentage are 5, 10, 25, 50, 65 and 80.

Fig. 6.26 shows the comparison of the size required to store the graphs over varying sizes and densities using adjacency matrix (AdjMat), quadtree (or Pure Quadtree denoted by PQT) and hybrid quadtree (HQT). From the plots it can be inferred that hybrid quadtree is better than the other two representations except when the density is around 50%. When the density is around 50% the adjacency matrix performs better for most of the cases.

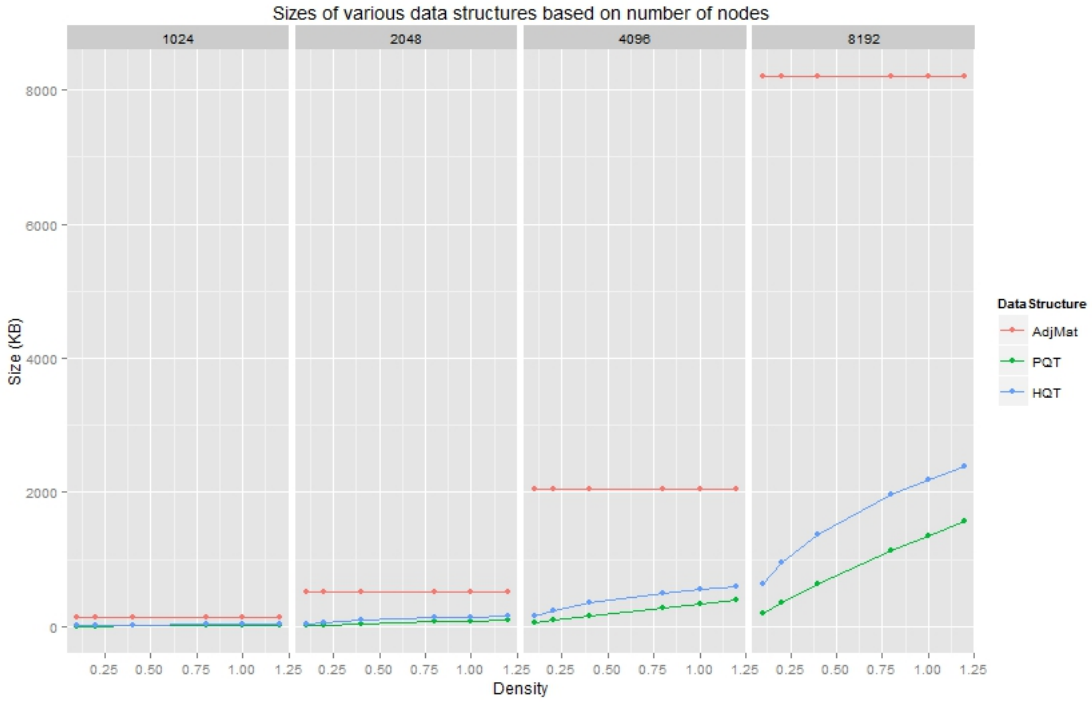


Figure 6.27: Data representation comparison for low densities

However, in case of real-world data representing online social networks and road networks, the graphs are sparse. Therefore, it is important to consider graphs with low densities and check the effectiveness of the proposed data structures. For low density graphs, the densities considered are 0.10, 0.20, 0.40, 0.80, 1.00 and 1.20. Fig. 6.27 the plots for the graphs of the same sizes and data structures as considered

before using low densities. From the plots it can be inferred that the quadtree (or pure quadtree) representation is the most efficient one; this is also partly because of the extra identifier used in the hybrid quadtree representations. For all the graph sizes, the adjacency matrix performs worse as expected for low densities.

6.4 Summary

In this chapter we study graph compression techniques and focus on quadtree representation of graphs. It is also shown how numbering of the nodes in graphs influence the space required for the quadtree. Special graphs are considered and analyzed. Techniques that modify graphs by adding and deleting edges also help in reducing the overall space requirements. Different size graphs are chosen over varying densities and sizes are compared by performing empirical analysis on the same. From the results it can be inferred that quadtree is indeed an effective compression technique for storing graphs specifically on the GPUs.

Chapter 7

Conclusion

7.1 Summary

In this dissertation, graph algorithms that are related to data analysis on large networks are studied using Graphics Processing Units (GPUs). In general, techniques employed to use GPUs to solve general purpose problems have been introduced and analyzed. The algorithms are designed and executed on different architectures of CUDA enabled GPUs using real-world data for analysis.

CUDA enabled GPUs with multiple Streaming Multi-processors (SMs) provide huge computation potential that can be exploited by applications using extensive multi-threading. The initial focus of the research has been on using the shared memory on the GPU because of its low memory access latency. However, since the shared memory is also limited in size as compared to the other levels in the memory hierarchy of the GPU, using efficient data structures are essential. With the use of advanced data structures as compared to the naive ones, larger graphs can be stored on the shared memory and computed on. Specifically, given a graph $G = (V, E)$ and an integer $k <= |V|$, both storage techniques and algorithms are introduced to count the number of a) connected subgraphs of size k, b) k cliques, and c) k independent sets, all of which can be exponential in number. Storing the entire graph in memory for computation is not always necessary. For specific problems, only a portion of the data is required. It is shown that using the breadth-first search (BFS) tree data structure, the graph can be separated into levels and only a part of the entire tree

is required for computations. But, if the graph is dense and has only a few levels in the BFS-tree, it might not be a good candidate to benefit from such techniques. However, most real world graphs are sparse and the results hold. Also, since a major part of the data can be excluded from calculations, test cases are also reduced by a significant number. It can be inferred from the experimental results that the fastest computation times are shown for graphs that are stored on the shared memory and the computations are performed using the BFS-tree information. The counting problems show a speedup by 5 times for smaller graphs and 10 times for larger graphs for implementations compared between CPU and GPU.

Transforming an algorithm into its parallel version is essential for achieving improvements using GPUs. With the added overhead of data transfer back and forth between the CPU and the GPU, enough SMs and GPU cores must be used to achieve significant speedup as compared to using the CPU alone. Also, it must be noted that in most cases a heterogeneous combination of CPU-GPU system is used to solve the problem being considered; preprocessing is done on the CPU and the parallel part of the problem is solved on the GPU. Identifying computations that can be performed independently is essential for the conversion of the algorithms and implementation into GPU kernels. Also, since accessing data from the memory can be a bottleneck, separating computations based on data access patterns is also significant.

The GPU architecture and CUDA programming environment, including the programming model and memory model is discussed in detail. Issues arising from shared memory bank conflicts are studied and methods to avoid the same are discussed. Also, since the domain of problems focus on real-world data, and it is huge in size, using the much larger global memory on the device is required. Since memory is at a premium, various data structures including both naive and advanced versions are introduced and analyzed. However, since the global memory has a much higher memory access latency as compared to the shared memory, using it without employing

any optimization principles reduces the effectiveness of the multicore architecture. Using primitives like memory access coalescing and avoiding partition camping are discussed; detailed description on how to exploit the primitives to improve the performance of the global memory is also included in the context of solving the triangle counting problem. From empirical results for solving the triangle counting problem, it can be concluded that the performance of the GPU increases by approximately 6-8% over the naive GPU implementation when making use of the available primitives.

Algorithms for different counting problems have been proposed in this dissertation. The general approach involves generating combinations of nodes for a given size, and checking if the specific property being studied holds in case of the induced sub-graph. Therefore, generating combinations to test for a specific property is an important part of the process. Exploiting the structure and characteristics of a BFS-tree representation of the graph being considered, a number of combinations can be excluded from calculations thereby reducing the number of combinations to be generated by a significant factor. Equal division of work among threads is important to exploit the potential benefits of multi-threading. Generating all combinations as part of preprocessing and dividing them equally among threads is not a suitable option for GPUs since storing the combinations itself would require significant amount of memory. Combinadics is a technique that can be used to generate combinations on the fly and out of sequence, and can therefore be executed independent of other data and in parallel. Therefore, generating combinations using combinadics is useful in this aspect and is shown in the context of the triangle counting algorithm.

Counting triangles has many important applications and has been studied here in detail. Real-world graphs are dynamic, and edges and nodes can be added or deleted over time. To incorporate such dynamic behavior, triangle completion and edge deletion problems have been considered. Finding exact solutions to problems on large graphs require a lot of computations. In such cases, approximate counting is a

viable and fast alternative. A heuristic for triangle completion problem is proposed and compared with the exact solution.

Online social networks and road networks, among others, are the domains that have been looked into while studying the graph problems. Since the sizes of the graphs are huge and computation on the same would be expensive, analyzing the structure of the data to check for dependency patterns that can partition the data into smaller sets is relevant. By performing analysis on the real-world graphs valuable insights are gained that conclude most of the graphs have many levels in the BFS-tree; the size of sets of adjacent levels can be accommodated in the GPU memory taking into account the separate components that exist within the data being considered.

Data transfer between the CPU and GPU memory is expensive. Also, with more data being present on the device memory, computations can be executed efficiently. Since the data being considered in all the problems belong to large graphs, being able to use efficient data structures to store as much information as possible is relevant. Graph compression techniques help address this issue. Quadtree representation of graphs are introduced and other methodologies discussed that can reduce the memory requirements for graphs. Empirical results show that using a hybrid data structure, which is a combination of quadtree and adjacency matrix is an effective compression technique.

7.2 Future Work

In this dissertation, graphs that fit into the GPU memory using efficient data structures have been considered. As an extension, graphs that do not fit on the GPU memory, and require multiple kernel calls and memory transfers to be computed on, could be addressed. Being able to compute on streaming graphs that are much larger in size, and need to be handled using external memory would be relevant.

Various optimization techniques have been studied to achieve better performance on the GPU based on graph algorithms. Another avenue to extend this work would be to explore the effects of such techniques on algorithms and data belonging to other domains.

The CUDA enabled GPU devices provide various hardware and software techniques to improve performance on the same. Data access from the shared memory can take advantage of a broadcast mechanism if certain criteria are met; also shared memory is divided into banks to avoid sequential reads and writes. However, broadcast mechanism is not available when accessing data from the global memory under any circumstances; also the number of partitions in the global memory, to facilitate concurrent access, is much less in number compared to the banks in shared memory. It would be interesting to study the effects if certain changes can be made to the model. Therefore, an extension to the current work could involve using a simulator to redesign specific aspects of the programming and memory model to predict the gain in performance if certain modifications can be incorporated.

Bibliography

Advanced Micro Devices (2013). AMD Accelerated Parallel Processing OpenCL Programming Guide.

Bader, D. A., & Madduri, K. (2008). SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IEEE International Symposium on Parallel and Distributed Processing* (pp. 1 –12).

Becchetti, L., Boldi, P., Castillo, C., & Gionis, A. (2008). Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining* KDD '08 (pp. 16–24). New York, NY, USA: ACM.

Benevenuto, F., Rodrigues, T., Cha, M., & Almeida, V. (2009). Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference* IMC '09 (pp. 49–62). New York, NY, USA: ACM.

Boldi, P., & Vigna, S. (2004). The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web* WWW '04 (pp. 595–602). New York, NY, USA: ACM.

Bordino, I., Donato, D., Gionis, A., & Leonardi, S. (2008). Mining Large Networks with Subgraph Counting. In *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on* (pp. 737 –742).

Borgelt, C., & Berthold, M. (2002). Mining molecular fragments: finding relevant substructures of molecules. In *Proceedings of IEEE International Conference on Data Mining, 2002.* (pp. 51–58).

Boyer, M., Skadron, K., & Weimer, W. (2008). Automated dynamic analysis of CUDA programs. In *Third Workshop on Software Tools for MultiCore Systems*.

Buckles, B. P., & Lybanon, M. (1977). Algorithm 515: Generation of a Vector from the Lexicographical Index [G6]. *ACM Transactions on Mathematical Software*, *3*, 180–182.

Buehrer, G., & Chellapilla, K. (2008). A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining* WSDM '08 (pp. 95–106). New York, NY, USA: ACM.

Buluç, A., Gilbert, J. R., & Budak, C. (2010). Solving path problems on the GPU. *Parallel Computing*, *36*, 241–253.

Chandran, L. S., Ibarra, L., Ruskey, F., & Sawada, J. (2003). Generating and characterizing the perfect elimination orderings of a chordal graph. *Theor. Comput. Sci.*, *307*, 303–317.

Chatterjee, A., Radhakrishnan, S., & Antonio, J. K. (2012). Counting Problems on Graphs: GPU Storage and Parallel Computing Techniques. In *IEEE International Symposium on Parallel and Distributed Processing Workshops, APDCM*.

Chatterjee, A., Radhakrishnan, S., & Antonio, J. K. (2013a). Data Structures and Algorithms for Counting Problems on Graphs using GPU. *International Journal of Networking and Computing (IJNC)*, *Vol. 3*, pages 264–288.

Chatterjee, A., Radhakrishnan, S., & Antonio, J. K. (2013b). On Analyzing Large Graphs Using GPUs. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (pp. 751–760). IEEE.

Chatterjee, A., Radhakrishnan, S., & Sekharan, C. N. (2014). Connecting the dots: Triangle completion and related problems on large data sets using GPUs. In *2014 IEEE International Big Data Workshop on High Performance Big Graph Data Management, Analysis, and Mining*. IEEE.

Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., & Raghavan, P. (2009). On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '09 (pp. 219–228). New York, NY, USA: ACM.

Chu, S., & Cheng, J. (2011). Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* KDD '11 (pp. 672–680). New York, NY, USA: ACM.

Deshpande, A. (2010). Graph Compression. https://www.cs.umd.edu/class/fall2010/cmsc828e/compression.pdf.

Eckstein, D. M. (1979). *BFS and biconnectivity*. Technical Report Iowa State University of Science and Technology, Department of Computer Science.

Facebook Statistics (2014). https://newsroom.fb.com/company-info/.

Feder, T., & Motwani, R. (1991). Clique Partitions, Graph Compression and Speeding-up Algorithms. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing* STOC '91 (pp. 123–133). New York, NY, USA: ACM.

Frishman, Y., & Tal, A. (2007). Multi-Level Graph Layout on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, *13*, 1310–1319.

Garland, M. (2008). Sparse matrix computations on manycore GPUs. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE* (pp. 2 –6).

Grabowski, J., & Wodecki, M. (2004). A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. In *Computers & Operations Research* (pp. 1891–1909).

Gustavson, F. G., Waśniewski, J., Dongarra, J. J., & Langou, J. (2010). Rectangular full packed format for cholesky's algorithm: factorization, solution, and inversion. *ACM Trans. Math. Softw.*, *37*, 18:1–18:21.

Harish, P., & Narayanan, P. J. (2007). Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proc. of the IEEE Intl Conf. on High Performance Computing, LNCS 4873* (pp. 197–208).

Hasan, K. S., Chatterjee, A., Radhakrishnan, S., & Antonio, J. K. (2014). Performance Prediction Model and Analysis for Compute-Intensive Tasks on GPUs. In *Network and Parallel Computing - 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings* (pp. 612–617).

Hong, S., Kim, S. K., Oguntebi, T., & Olukotun, K. (2011). Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* PPoPP '11 (pp. 267–276). New York, NY, USA: ACM.

Itokawa, T., Tada, A., & Migita, M. (2007). Parallel Algorithm for Finding the Minimum Edges to Make a Disconnected Directed Acyclic Graph Strongly Connected. In *Innovative Computing, Information and Control, 2007. ICICIC '07. Second International Conference on* (p. 131).

Katz, G. J., & Kider, J. T., Jr (2008). All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (pp. 47–55). Eurographics Association.

Leskovec, J., Adamic, L. A., & Huberman, B. A. (2007). The dynamics of viral marketing. *ACM Trans. Web*, *1*.

Leskovec, J., Lang, K., Dasgupta, A., & Mahoney, M. (2009). Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, *Vol. 6*, 29–123.

LinkedIn Press Center (2014). http://press.linkedin.com/about.

McCaffrey, J. (2004). Generating the m^{th} Lexicographical Element of a Mathematical Combination. http://msdn.microsoft.com/en-us/library/aa289166(v=vs.71).aspx.

Mifsud, C. J. (1963). Algorithm 154: Combination in Lexicographical Order. *Communications of the ACM*, *6*, 103–105.

Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., & Bhattacharjee, B. (2007). Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement* IMC '07 (pp. 29–42). New York, NY, USA: ACM.

Mislove, A., Viswanath, B., Gummadi, K. P., & Druschel, P. (2010). You are who you know: Inferring user profiles in online social networks. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining* WSDM '10 (pp. 251–260). New York, NY, USA: ACM.

NVIDIA Corporation (2010). NVIDIA CUDA C Programming Guide, Version 3.2.

Randall, K. H., Stata, R., Wiener, J. L., & Wickremesinghe, R. G. (2002). The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference* DCC '02 (pp. 122–). Washington, DC, USA: IEEE Computer Society.

Ruetsch, G., & Micikevicius, P. (2009). Optimizing matrix transpose in CUDA. *NVIDIA Technical Report*, .

Samet, H. (1985). Using quadtrees to represent spatial data. In H. Freeman, & G. Pieroni (Eds.), *Computer Architectures for Spatially Distributed Data* (pp. 229–247). Springer Berlin Heidelberg volume 18 of *NATO ASI Series*.

Stanford Network Analysis Project (2011). Stanford Large Network Data Collection. https://snap.stanford.edu/data/index.html.

Sulewski, D., Edelkamp, S., & Kissmann, P. (2011). Exploiting the Computational Power of the Graphics Card: Optimal State Space Planning on the GPU. In *21st International Conference on Automated Planning and Scheduling*.

Suri, S., & Vassilvitskii, S. (2011). Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web* WWW '11 (pp. 607–614). ACM.

Tsourakakis, C. E., Kang, U., Miller, G. L., & Faloutsos, C. (2009). DOULION: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* KDD '09 (pp. 837–846). New York, NY, USA: ACM.

Twitter Statistics (2014). Twitter usage and company facts. https://about.twitter.com/company.

Vineet, V., Harish, P., Patidar, S., & Narayanan, P. J. (2009). Fast minimum spanning tree for large graphs on the GPU. In *High Performance Graphics'09* (pp. 167–171).

Viswanath, B., Mislove, A., Cha, M., & Gummadi, K. P. (2009). On the evolution of user interaction in facebook. In *Proceedings of the 2Nd ACM Workshop on Online Social Networks* WOSN '09 (pp. 37–42). New York, NY, USA: ACM.

Yang, Y., Xiang, P., Kong, J., & Zhou, H. (2010a). A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*.

Yang, Y., Xiang, P., Kong, J., & Zhou, H. (2010b). A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* PLDI '10 (pp. 86–97). New York, NY, USA: ACM.

Yu, H., Kaminsky, M., Gibbons, P. B., & Flaxman, A. (2006). SybilGuard: Defending Against Sybil Attacks via Social Networks. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications* SIGCOMM '06 (pp. 267–278). New York, NY, USA: ACM.

Zhao, Z., Wang, G., Butt, A. R., Khan, M., Kumar, V. S. A., & Marathe, M. V. (2012). SAHAD: Subgraph analysis in massive networks using Hadoop. In *2012 IEEE 26th International Parallel Distributed Processing Symposium* (pp. 390–401).

Appendix

Acronyms

AL-AL Adjacency List Using Array of Linked Lists

AL-AA Adjacency List Using Array of Arrays

AL-ALL Adjacency List Using Array Implementation of Linked Lists

AL-EG Adjacency List with Edges Grouped

AMD Advanced Micro Devices

API Application Programming Interface

BFS Breadth-First Search

B-S-UTM Balanced Strictly Upper Triangular Matrix

CPU Central Processing Unit

CRCW Concurrent Read Concurrent Write

CREW Concurrent Read Exclusive Write

CRN California Road Network

CUDA Compute Unified Device Architecture

EEN Enron Email Network

EREW Exclusive Read Exclusive Write

FSC Facebook Social Circles

GPU Graphics Processing Unit

GPGPU General Purpose Graphics Processing Unit

HQT Hybrid Quadtree

ITG Internet Topology Graph

MPI Message Passing Interface

OSN Online Social Networks

PAR Parent Array Representation

PEO Perfect Elimination Ordering

PRAM Parallel Random Access Machine

PQT Pure Quadtree

PRN Pennsylvania Road Network

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SNAP Stanford Network Analysis Project

SPMD Single Program Multiple Data

SM Streaming Multi-processor

S-UTM Strictly Upper Triangular Matrix

TRN Texas Road Network

UTM Upper Triangular Matrix